

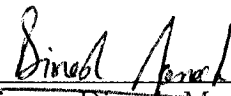
Efficient Visibility-Based Algorithms for Interactive Walkthrough, Shadow Generation, and Collision Detection

by
Naga K. Govindaraju

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2004

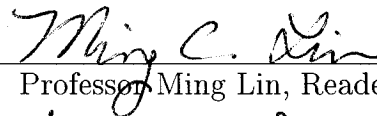
Approved by:



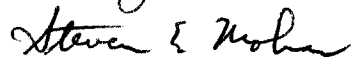
Professor Dinesh Manocha, Advisor



Professor Anselmo Lastra, Reader



Professor Ming Lin, Reader



Professor Steven Molnar, Reader

© 2004
Naga K. Govindaraju
ALL RIGHTS RESERVED

ABSTRACT

NAGA K. GOVINDARAJU: Efficient Visibility-Based Algorithms for Interactive Walkthrough, Shadow Generation, and Collision Detection.
(Under the direction of Professor Dinesh Manocha.)

We present novel visibility-based algorithms for solving three problems: interactive display, shadow generation, and collision detection. We use the visibility capabilities available on graphics processing units (GPUs) in order to perform visibility computations, and we apply the results to complex 3-D environments.

We present a real-time visibility culling algorithm that reduces the number of rendered primitives by culling portions of a complex 3-D environment that are not visible to the user. We introduce an occlusion switch to perform visibility computations. Each occlusion switch consists of two GPUs, and we use these GPUs either for computing an occlusion representation or for culling away primitives from a given view point. Moreover, we switch the roles of each GPU across successive frames. The visible primitives are rendered on a separate GPU or are used for generating hard-edged umbral shadows from a moving light source. We use our visibility culling algorithm for computing the potential shadow receivers and shadow casters from the eye and light sources, respectively. We further reduce their size using a novel cross-culling algorithm.

We present a novel visibility-based algorithm for reducing the number of pair-wise interference tests among multiple deformable and breaking objects in a large environment. Our collision detection algorithm computes a *potentially colliding set (PCS)* using image-space occlusion queries. Our algorithm involves no precomputation and proceeds in multiple stages: PCS computation at an object level and PCS computation at a subobject level, followed by exact collision detection. We use a linear-time two-pass rendering algorithm for computing each PCS efficiently. Our collision-pruning algorithm can also compute self-collisions in general deformable models. Further, we overcome the

sampling and precision problems in our pruning algorithm by fattening the triangles sufficiently in PCS. We show that the *Minkowski sum* of each primitive with a sphere provides a conservative bound for performing reliable 2.5-D overlap tests using GPUs. In contrast to prior GPU-based collision detection algorithms, our algorithm guarantees that no collisions will be missed due to limited frame-buffer precision or quantization errors during rasterization.

We have implemented our visibility-based algorithms on PCs with a commodity graphics processor such as the NVIDIA GeForce FX 5900. We highlight their performance on complex 3-D environments composed of millions of polygons. In practice, we are able to achieve interactive frame rates of 10 - 20 frames per second in these experiments.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dinesh Manocha for his excellent guidance and support throughout the course of the dissertation. Thanks to Prof. Ming Lin for the feedback and collaboration in the collision detection projects. I would like to express my gratitude to Dr. Steven Molnar for providing an opportunity to work in the NVIDIA architecture group and for being a great mentor. The internship provided a valuable learning experience in understanding the design and functionality of graphics processors. Thanks to the rest of my dissertation committee for the feedback and discussions: Profs. Frederick Brooks and Anselmo Lastra.

The research projects reported in this dissertation involved the efforts of several student collaborators in UNC GAMMA and Walkthrough research groups. I appreciate their help and support. In this regard, I would like to acknowledge the efforts of Brandon Lloyd, Stephane Redon, Avneesh Sud, and Sungeui Yoon in developing the research systems. Thanks to the UNC Computer Science graduate students for making my stay comfortable, and to my friends who have always encouraged me during the difficult times. I am thankful to the many UNC Computer Science faculty for being considerate towards my efforts in balancing the demanding needs of research and coursework.

The technical support center of the UNC Computer Science department was extremely helpful in maintaining the group machines. In this regard, I would like to mention the efforts of Charlie Bauserman, Mike Carter and Mike Stone for their help. I am thankful to NVIDIA Corporation for providing both software and hardware support which have been extremely useful in our research projects. Paul Keller and Stephen

Ehmann of NVIDIA Corporation offered excellent driver support during critical deadlines and I am grateful to them.

The research projects were funded in parts by Army Research Office (grant DAAD19-99-1-0162), National Science Foundation (NSF awards ACI-9876914, ACI-9876914, IIS-982167, and ACI-0118743), Office of Naval Research (contracts N00014-97-1-0631, N00014-01-1-0067, and N00014-01-1-0496), Department of Energy ASCI grant, and Intel Corporation.

I am grateful to my parents and family for their love, encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES	xii
LIST OF FIGURES	xiii
1 Introduction	1
1.1 Visibility from a Point	2
1.2 Visibility Algorithms	3
1.3 Goals	5
1.4 Prior Work and Challenges	7
1.4.1 Visibility for Rendering Algorithms	7
1.4.2 Visibility for Shadow Algorithms	11
1.4.3 Visibility for Collision Detection	12
1.5 Graphics Processors	14
1.6 Thesis Statement	21
1.7 New Results	21
1.8 Organization	25
2 Related Work	26
2.1 Large-Model Rendering	26
2.1.1 Polygonal Simplification	27
2.1.2 Image-Based Representations	30

2.1.3	Visibility Culling	32
2.1.4	Parallel Rendering	40
2.2	Shadow Generation	40
2.2.1	Image-Precision Methods	40
2.2.2	Object-Precision Methods	42
2.2.3	Hybrid Approaches	44
2.3	Interference Detection	44
2.3.1	Object-Space Algorithms	45
2.3.2	Image-Space Algorithms	46
2.3.3	Hybrid Algorithms	48
3	Visibility Computations: Interactive Walkthroughs	49
3.1	Introduction	49
3.2	Interactive Occlusion Culling	51
3.2.1	Occlusion Representation and Culling	51
3.2.2	Occlusion Switch	53
3.2.3	Culling Algorithm	54
3.2.4	Incremental Transmission	54
3.2.5	Bandwidth Requirements	55
3.3	Scene Representation	56
3.3.1	Unified Scene Hierarchy	57
3.3.2	Hierarchy Generation	58
3.3.3	HLOD Generation	62
3.3.4	HLODs as Hierarchical Occluders	63
3.4	Interactive Display	63

3.4.1	Culling Algorithm	64
3.4.2	Occluder-Representation Generation	65
3.4.3	Occlusion-Switch Algorithm	65
3.4.4	Render Visible Geometry	65
3.4.5	Incremental Traversal and Front Tracking	66
3.4.6	Optimizations	69
3.4.7	Design Issues	70
3.5	Implementation and Performance	72
3.5.1	Preprocessing	72
3.5.2	Run-time System	73
3.5.3	Bandwidth Estimates	76
3.6	Analysis	76
3.6.1	Comparison with Earlier Approaches	78
3.6.2	Limitations	79
4	Visibility Computations: Interactive Shadow Generation	83
4.1	Introduction	83
4.2	Shadow Culling	85
4.2.1	LOD-based Interactive PVS Computation	86
4.2.2	Cross-Culling	91
4.3	Applications	92
4.3.1	Hybrid Shadow-Generation Algorithm	93
4.3.2	CC Shadow Volumes	94
4.4	Implementation and Performance	94
4.4.1	Implementation	95

4.4.2	Performance	96
4.5	Analysis and Limitations	99
4.5.1	Interactive Performance and Load Balancing	99
4.5.2	Comparison with Other Approaches	100
4.5.3	Limitations	101
5	Visibility Computations: Interactive Collision Detection	105
5.1	Introduction	105
5.2	Collision Detection Using Visibility Queries	108
5.2.1	Potentially Colliding Set (PCS)	108
5.2.2	Visibility-Based Pruning	109
5.2.3	Localizing the Overlapping Features	111
5.2.4	Collision Detection	112
5.3	Self-Collision Culling using GPUs	112
5.4	Reliable Culling Using GPUs	115
5.4.1	Sampling Errors	116
5.4.2	Reliable VO Queries	117
5.4.3	Collision culling	119
5.5	Interactive Collision Detection	120
5.5.1	Pruning Algorithm	120
5.5.2	Visibility Queries	121
5.5.3	Multiple-Level Pruning	122
5.5.4	CULLIDE	123
5.5.5	S-CULLIDE	124
5.5.6	FAR	125

5.5.7	Optimizations	129
5.6	Implementation	130
5.6.1	CULLIDE	130
5.6.2	S-CULLIDE	131
5.6.3	FAR	132
5.7	Analysis and Limitations	133
5.7.1	Performance Analysis	133
5.7.2	Pruning Efficiency	137
5.7.3	Precision	138
5.7.4	Comparison with Other Approaches	139
5.7.5	Limitations	141
6	Conclusions	144
6.1	Walkthroughs	146
6.1.1	Future Work	147
6.2	Shadow Generation	147
6.2.1	Future Work	148
6.3	Collision Detection	148
6.3.1	Future Work	149
	BIBLIOGRAPHY	152

LIST OF TABLES

3.1	<i>Average frame rates obtained by different acceleration techniques over the sample path. FPS = Frames Per Second, PP = Power Plant model, DE = Double Eagle Tanker model, B-777 = Boeing 777 model</i>	74
3.2	<i>Comparison of the number of polygons rendered by the two implementations to the actual number of visible polygons. PP = Power Plant model, DE = Double Eagle Tanker model</i>	74
3.3	<i>Comparison of the number of objects rendered by the two implementations to the actual number of visible objects. PP = Power Plant model, DE = Double Eagle Tanker model</i>	75

LIST OF FIGURES

1.1	<i>Complex piping in the Double Eagle Tanker model: This figure shows an example of a complex scene with no large occluders or portals. The scene is composed of 82 million polygons. Model Courtesy: Newport News Shipping company.</i>	4
1.2	<i>Complex piping in the Power Plant model: This figure shows the dense piping structures present in the furnace room of the Power Plant model. Observe that each pipe occludes a very small portion of the furnace room, whereas a group of spatially proximate pipes can occlude most portions of the scene. The scene is composed of 13 million polygons.</i>	6
1.3	<i>Different culling techniques on a scene shown in (i): view-frustum culling in (ii), back-face culling in (iii), and occlusion culling in (iv).</i>	8
1.4	<i>Double Eagle Tanker model: This model is composed of 82 million polygons and exhibits high visual and depth complexity.</i>	9
1.5	<i>This figure shows a simple primitive projected on the screen and its screen-space bounding rectangle. Observe that the screen-space bounding rectangle occupies a much larger screen space in comparison to the projected primitive and can be quite conservative for performing visibility computations.</i>	10
1.6	<i>This figure shows a snapshot of a dynamic simulation consisting of many objects in close proximity. These close-proximity scenarios are challenging for most interactive collision detection algorithms. In this simulation, as time progresses, there are more than a thousand objects in close proximity. These objects in the scene together are composed of 150K polygons.</i>	13
1.7	<i>Super Moore's law : This figure shows the performance growth rates of GPU, CPU, and AGP bandwidth between GPU and CPU. GPU's computational power has been progressing at a rate more than Moore's law for CPUs. Courtesy Anselmo Las-tra, UNC</i>	14
1.8	<i>Graphics architectural pipeline overview: This figure shows the various units of a modern GPU. Each unit is designed for performing a specific operation efficiently.</i>	16

1.9	<i>A snapshot generated from the application of our hybrid shadow-generation algorithm to the Power Plant model (12.7 million triangles). Our interactive algorithm is able to generate shadows at an average frame rate of 10 fps using three PCs, each with an NVIDIA GeForce 4 graphics card.</i>	23
3.1	<i>This figure shows a view of the Boeing 777 model. The model is designed using more than 470 million polygons.</i>	49
3.2	<i>System Architecture: Each color represents a separate GPU. Note that GPU₁ and GPU₂ switch their roles each frame with one performing hardware culling and other rendering occluders. GPU₃ is used as a display client.</i>	52
3.3	<i>Our clustering and partitioning process applied to a 2-D example. Each different color represents a different object at the end of a stage. (a) The model's original objects. This object distribution captures a number of features common in CAD models in which objects are defined by function rather than by location. (b) The initial partitioning stage splits objects with large bounding boxes. This prevents objects like 3, whose initial bounding box intersects most of the others, from causing clustering to generate just one large cluster. (c) After clustering, the group of the small objects around 1 have all been merged to form 1*. The row of objects, 2, has been merged into one cluster, 2*, as well, but one which has a poor aspect ratio. (d) The final partition splits 2* into two separate objects.</i>	56
3.4	<i>The image on the left shows the application of the partitioning and clustering algorithm to the Power Plant model. The middle image shows the original objects in the Double Eagle tanker model with different colors. The right image shows the application of the clustering algorithm to the same model. Each cluster is shown with a different color.</i>	60
3.5	<i>System Overview: Each color represents a separate GPU, with GPU₁ and GPU₂ forming a switch and GPU₃ as the display client. GPU₁ and GPU₂ each have a camera-receiver thread; receive camera parameters when the client transmits them due to user's motion; and store those in a camera buffer of size one. The GPU performing OR takes the latest camera from this thread as the camera position for the next frame. Notice that, in this design, the GPU performing HC exhibits no latency in receiving the camera parameters.</i>	67

3.6	<i>This figure shows a cut defined by the traversal of the scene graph using our culling algorithm. The cut is further decomposed into visible and occluded fronts. Each front is represented using the following colors: orange: visible front; and gray: occluded front.</i>	68
3.7	<i>Comparison of the number of nodes transmitted with and without incremental transmission (described in Section 3.4.5) for a sample path on Double Eagle Tanker model. Using incremental transmission, we observe an average reduction of 93% in the number of nodes transmitted between the GPUs.</i>	72
3.8	<i>Frame-rate comparison between SWITCH and Distributed GigaWalk at 1024×1024 screen resolution and 15 pixels of error on the Boeing model.</i>	80
3.9	<i>Frame-rate comparison among SWITCH, GigaWalk and Distributed GigaWalk at 1024×1024 screen resolution. We obtain 2–3 times improvement in the frame rate as compared to Distributed GigaWalk and GigaWalk.</i>	81
3.10	<i>Double Eagle Tanker: Comparison of exact-visibility computation with SWITCH and GigaWalk at 20 pixels of error at 1024×1024 screen resolution. SWITCH is able to perform more culling than GigaWalk; however, it renders one order of magnitude more triangles or twice the number of objects as compared to exact visibility.</i>	81
3.11	<i>Performance of the occlusion-switch algorithm on the Double Eagle Tanker model: This environment consists of more than 82 million triangles, and our algorithm renders it at 9–15 fps on a cluster of 3 PCs, each consisting of an NVIDIA GeForce 4 GPU. Occlusion switch culls away most occluded portions of the model and renders around 200K polygons in the view shown. Objects are rendered in the following colors: visible: yellow; view-frustum culled: violet; and occlusion-culled: orange.</i>	82
3.12	<i>Performance of occlusion switch on complex CAD models: Both models are rendered at 1024×1024 screen resolution using NVIDIA GeForce 4 cards.</i>	82

4.1	<i>The left image shows a snapshot generated from the application of our shadow culling algorithms and a hybrid shadow-generation technique to the Power Plant model (12.7M triangles). The middle image shows a different viewpoint generated using perspective shadow maps. Notice the aliasing artifacts. The right image highlights the sharper boundaries of the shadows generated by our interactive algorithm from the same viewpoint.</i>	83
4.2	<i>Visibility Computation for Shadow Generation: This figure shows a simple scene with a light source placed above a sphere and a floor. Shadows are regions on the floor and sphere that are visible to the eye, but not visible to the light.</i>	86
4.3	<i>This figure shows a cut defined by the traversal of the scene graph using our improved culling algorithm. The cut is composed of visible and occluded nodes. Each node in the hierarchy is composed of several HLODs and each HLOD is further decomposed into multiple subobjects to improve the culling efficiency of our algorithm.</i>	87
4.4	<i>Self-shadowing artifacts due to a naive LOD-selection algorithm. We correct this problem by using the same LOD parameter for an object when computing PVS_E and PVS_L.</i>	88
4.5	<i>The effect of increasing the LOD-error threshold in the Double Eagle Tanker model. These images have been generated with 0 (left), 10 (middle), and 20 (right) pixels of error.</i>	89
4.6	<i>Overview of our hybrid approach showing the four stages of the algorithm and the intermediate computations.</i>	92
4.7	<i>Architecture of the Process-Parallel Algorithm: This figure shows the components of our hybrid shadow generation algorithm. Each color represents a separate graphics processor or CPU.</i>	93
4.8	<i>A snapshot generated from an application of our interactive shadow generation algorithm to the house model. The model has about 1.3M triangles. No LODs were used.</i>	95
4.9	<i>A sequence generated by a light source moving over the Power Plant away from the viewer. Our algorithm can generate shadows at 10 frames per second on average for this model.</i>	96

4.10	<i>A snapshot of the tanker model rendered using our system. The tanker has more than 82 million triangles. This view highlights the shadows generated by the long and thin pipes on the deck.</i>	98
4.11	<i>Frame rates obtained for each model.</i>	103
4.12	<i>Performance of the culling techniques. OC_E refers to the number of triangles after object culling. The PVSs are obtained by performing subobject culling. There is a reduction of almost an order of magnitude in the size of PVS_E as compared to that of OC_E. After cross-culling, the sizes of the shadow casters (SC) and shadow receivers used for calculating shadow boundaries are each on the order of a few thousand.</i>	104
4.13	<i>Comparison of shadows generated by uniform shadow maps (left), perspective shadow maps (middle), and our hybrid algorithm (right). Each image also includes a zoomed view of the shadow boundaries on the top left corner. Perspective shadow maps reduce some of the aliasing artifacts as compared to uniform shadow maps; however, they are unable to generate sharp shadows in many scenarios.</i>	104
5.1	<i>Tree with Falling Leaves: In this scene, leaves fall from the tree and undergo nonrigid motion. They collide with other leaves and branches. The environment consists of more than 40K triangles and 150 leaves. Our GPU-based reliable collision detection algorithm, FAR, can compute all the collisions in about 35 msec per time step on a PC with 2.8 GHz Pentium IV CPU and NVIDIA GeForce FX 5950 Ultra GPU.</i>	105
5.2	<i>Potentially Colliding Set: In this viewpoint, two of the four objects are in close proximity and belong to the potentially colliding set.</i>	106
5.3	<i>In this figure, the two objects are not colliding. Using View 1, we determine a separating surface with unit depth complexity along the view and conclude from the existence of such a surface that the objects are not colliding. This surface's existence is a sufficient but not a necessary condition. Observe that in View 2, there does not exist a separating surface with unit depth complexity but the objects are not interfering.</i>	109
5.4	<i>System Architecture: The overall pipeline of the collision detection algorithm for large environments</i>	111

5.5	<i>The left image shows an object composed of triangles with shared edges and vertices. The right image shows the self-intersecting triangles in the object. Observe that these self-intersecting triangles do not share an edge or a vertex.</i>	113
5.6	<i>Sampling Errors: Q is a point on the line of intersection between two triangles in 3-D. The left figure highlights the orthographic projection of Q in the screen space. The intersection of the two triangles does not contain the center of the pixel (C), and therefore, we can miss a collision between the triangles. Q^B is the Minkowski sum of Q and an axis-aligned bounding box (B) centered at the origin with dimension p. Q^B translates B to the point Q. During rasterization, the projection of Q^B samples the center of the pixel and generates at least two fragments that bound the depth of Q.</i>	118
5.7	<i>This image shows an object with three triangles and its bounding-offset representation (UOBB) in wireframe. The UOBB is represented as the union of the OBBs of each triangle. In practice, this bounding offset is a tight-fitting bounding volume and is used for culling.</i>	127
5.8	<i>Results of our self-collision algorithm on a cloth simulation where a cloth consisting of 20K triangles drapes around a sphere. Figure (a) shows a snapshot of the simulation. Figure (b) shows the triangles in PSCS.</i>	131
5.9	<i>Breaking-Object Scene: In this simulation, the bunny model falls on the dragon which eventually breaks into hundreds of pieces. FAR computes collisions among the new pieces of small objects introduced into the environment and takes 30 to 60 msec per frame.</i>	133
5.10	<i>Number of Objects vs. Average Collision-Pruning Time using CULLIDE: This graph highlights the relationship between the number of objects in the scene and the average collision-pruning time (object pruning and subobject/triangle pruning). Each object is composed of 200 triangles. The graph indicates that the collision-pruning time is linear to the number of objects.</i>	134

5.11	<i>Polygons per Object vs. Average Collision-Query Time using CULLIDE: This graph shows the linear relationship between the number of polygons per object and the average collision-pruning time. This scene is composed of 100 deforming cylinders and has a density of 1 - 2%. The collision-pruning time is averaged over 500 frames and at an image-space resolution of 800×800.</i>	135
5.12	<i>Image-Space Resolution vs. Average Collision-Query Time using CULLIDE: This graph indicates the linear relationship between the screen resolution and the average collision-query time. The scene consists of 100 deformable cylinders and each object is composed of 200 triangles.</i>	136
5.13	<i>Relative Culling Performance on Breaking-Objects Scene: This graph highlights the improved culling performance of our algorithm as compared to a CPU-based culling algorithm (I-COLLIDE) that uses AABBs (axis-aligned bounding boxes) to cull away non-overlapping pairs. FAR reports 6.9 times fewer pairs over the entire simulation.</i>	137
5.14	<i>Reliable interference computation: This image highlights the intersection set between two bunnies, each with 68K triangles. The top right image (b) shows the output of FAR and the top left image (a) highlights the output of CULLIDE running at a resolution of 1400×1400. CULLIDE misses many collisions due to the viewport resolution and sampling errors.</i>	139
5.15	<i>Snapshots of Simulations on Three Complex Environments: CULLIDE takes 4, 8, and 40ms respectively on each benchmark in order to perform collision queries on a GeForce FX 5800 Ultra with an image resolution of 800×800.</i>	142
5.16	<i>Environment with Breakable Objects: As the bunny (with 35K triangles) falls through the dragon (with 250K), the number of objects in the scene (shown with a yellow outline) and the triangle count within each object change. CULLIDE computes all the overlapping triangles during each frame. The average collision-query time is 35 ms per frame.</i>	143

Chapter 1

Introduction

Visibility computations are fundamental in computer graphics, computer vision, robotics, computational geometry, etc. They have been researched extensively, and several visibility algorithms have been designed. In a recent survey (Bittner and Wonka, 2003), visibility algorithms are broadly classified based on the problems as follows:

1. visibility along a line
2. visibility from a point
3. visibility from a line segment
4. visibility from a polygon
5. visibility from a region
6. global visibility

Visibility computations require global processing and are often considered difficult and expensive in complex environments¹ (El-Sana et al., 2001). In this thesis, we restrict our focus to *visibility from a point* algorithms and apply them for solving three geometric problems on complex environments:

¹In this thesis, we refer to complex environments as scenes composed of several hundreds of thousands of geometric primitives.

1. **Interactive Walkthroughs:** Render the scene interactively as the user explores the environment.
2. **Shadow Generation:** Generate shadows at interactive rates in a scene with a moving *point* light source and a moving user.
3. **Collision Detection:** Compute colliding triangles interactively in a scene composed of dynamic objects.

In this chapter, we provide our definition of visibility from a point and the background for such a formulation. We briefly outline the prior work on the above three problems. We give an overview of commodity graphics processors and their features for performing visibility computations. We describe our approach and summarize our key results.

1.1 Visibility from a Point

The term *visible surface determination from a point* in computer graphics often relates to the problem of *hidden surface removal* (HSR) (Foley et al., 1990). Often, HSR algorithms remove hidden portions of a surface by shooting rays at the surface and computing the first point on the surface that intersects the ray. This formulation may not be useful in many visualization applications in which a user may wish to observe portions of a surface whose attribute values satisfy a constraint. These attributes include color, projected area in the view, and distance from the viewer. Therefore, we define *visibility from a point* as a *function* that operates on the attributes of geometric primitives and computes portions of these primitives satisfying the function.

Graphics processors are well designed for visualizing surfaces from a view position, and therefore the design of graphics pipeline has focused on defining the concepts of visibility and occlusion from a point for catering to the needs of rendering. Graphics architects have defined visibility as a composition of functions applied to the different

portions of a primitive, and we refer to these functions as visibility functions. The visibility functions operate on the attributes of a primitive, perform comparisons and provide a binary outcome. Each function is performed as a test at a stage in the pipeline. The pipeline is flexible, allowing any of these tests to be enabled or disabled. The tests are user-specified. Examples of these tests include *Alpha test*, *Stencil test*, and *Depth test*. Portions of a primitive that pass all these tests are considered visible. With this formulation, several geometric problems can be solved efficiently. Our definition of *visibility from a point* is based on the above formulation.

The above definition of visibility from a point allows for many interesting visualizations of geometric primitives based on their attributes. For example, a user may be interested in visualizing the portions of a surface that are farthest from a view position, or points on a surface with the alpha component of color greater than 0.5, or points on a surface restricted to a portion of the view. These visualizations are useful in many scientific applications related to computer graphics and computational geometry. One such application is walkthrough of virtual environments. Moreover, the definition exploits the capabilities of graphics processors in performing visibility computations efficiently. We will discuss these capabilities later, in Section 1.5.

1.2 Visibility Algorithms

Several visibility algorithms have been proposed for performing walkthroughs, generating shadows, and computing interferences. These algorithms can be broadly classified into three categories:

- **Exact visibility** techniques compute only the primitives that are visible to the user. This set of primitives comprises the exact visible set. Often, it is difficult to compute exact visible sets in real-time for complex general environments consisting of millions of polygons.

- **Approximate visibility** techniques compute most of the visible primitives but may miss some. It is difficult to guarantee quality or accuracy of computations using these techniques.
- **Conservative visibility** techniques compute a set of potentially visible primitives. The potentially visible set includes the primitives in the exact visible set and in addition, may include a few occluded primitives.

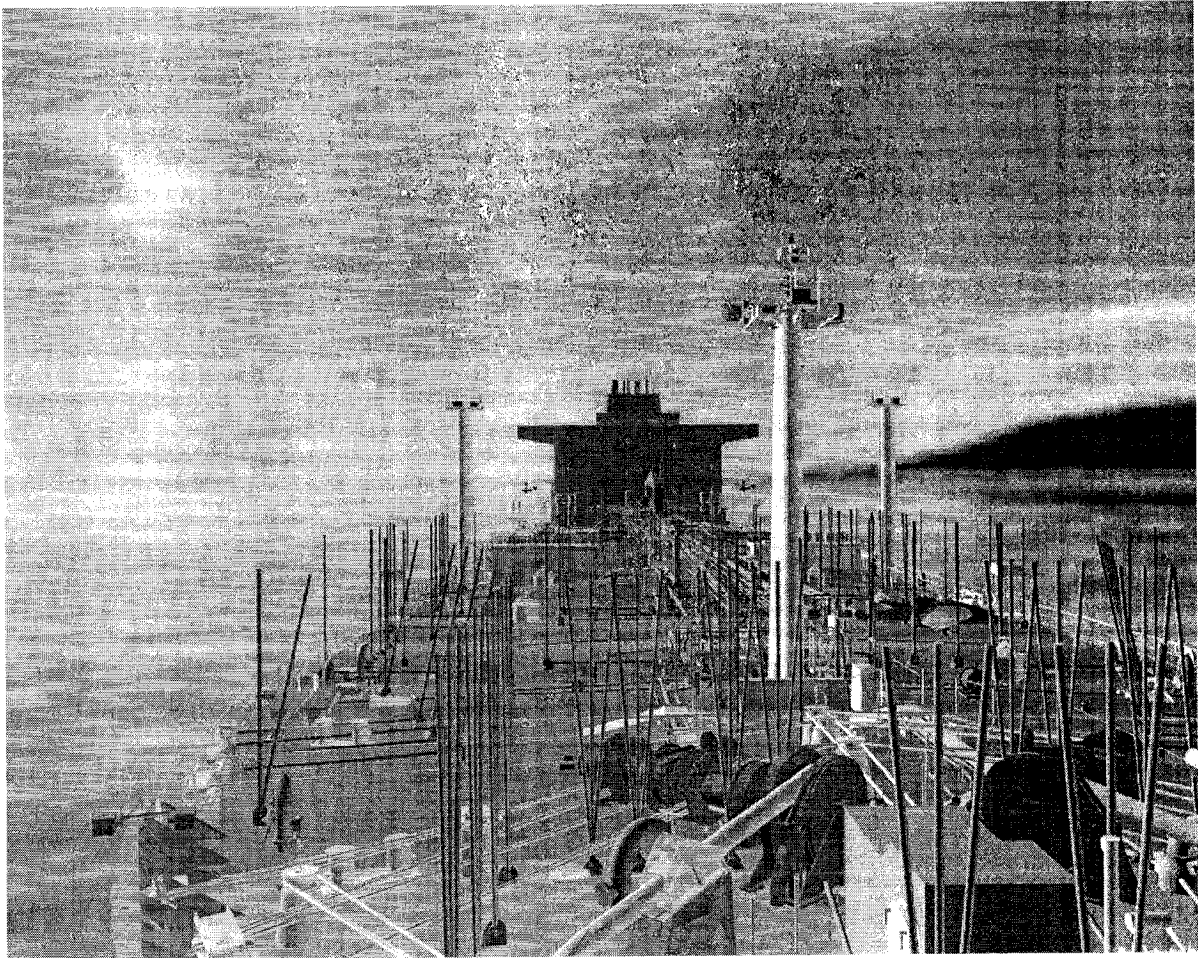


Figure 1.1: *Complex piping in the Double Eagle Tanker model: This figure shows an example of a complex scene with no large occluders or portals. The scene is composed of 82 million polygons. Model Courtesy: Newport News Shipping company.*

Visibility algorithms could be further classified as follows:

- **Image-precision algorithms** perform culling by using discrete representations as an image. As objects are rendered, portions of the screen are filled and used

for culling other portions of the scene. Image-precision algorithms are simple to implement and provide the advantage of occluder fusion ². In practice, they work well on general environments.

- **Object-precision algorithms** use 3-D objects for visibility computations (Cohen-Or et al., 2001a). They rely on the identification of large occluders or cells and portals. For general environments, however, it is difficult to identify large occluders or portals. Also, object-precision methods are usually less effective in performing occluder fusion (Cohen-Or et al., 2001a).
- **Hybrid algorithms** use a combination of object-precision and image-precision approaches (e.g., Hierarchical Z-Buffer (Greene et al., 1993)).

1.3 Goals

In this section, we present goals of our research. We highlight the challenges faced by prior algorithms in achieving these goals and the design decisions used within our algorithms.

The motivation for our research is studying the applications of visibility for performing interactive walkthroughs, shadow generation and collision detection in complex environments.

Goals: Our research focuses on achieving the following goals:

1. **Performance:** It is a key requirement for many interactive rendering applications. Our primary goal is to design algorithms capable of achieving interactive rendering rates (10 - 20 frames per second or more) on commodity graphics systems.
2. **Complex Environments:** Our goal is to handle models with high visual complexity and polygon count. In our rendering applications, we apply algorithms

²Occluder fusion refers to the effect of obtaining combined occlusion due to multiple occluders.

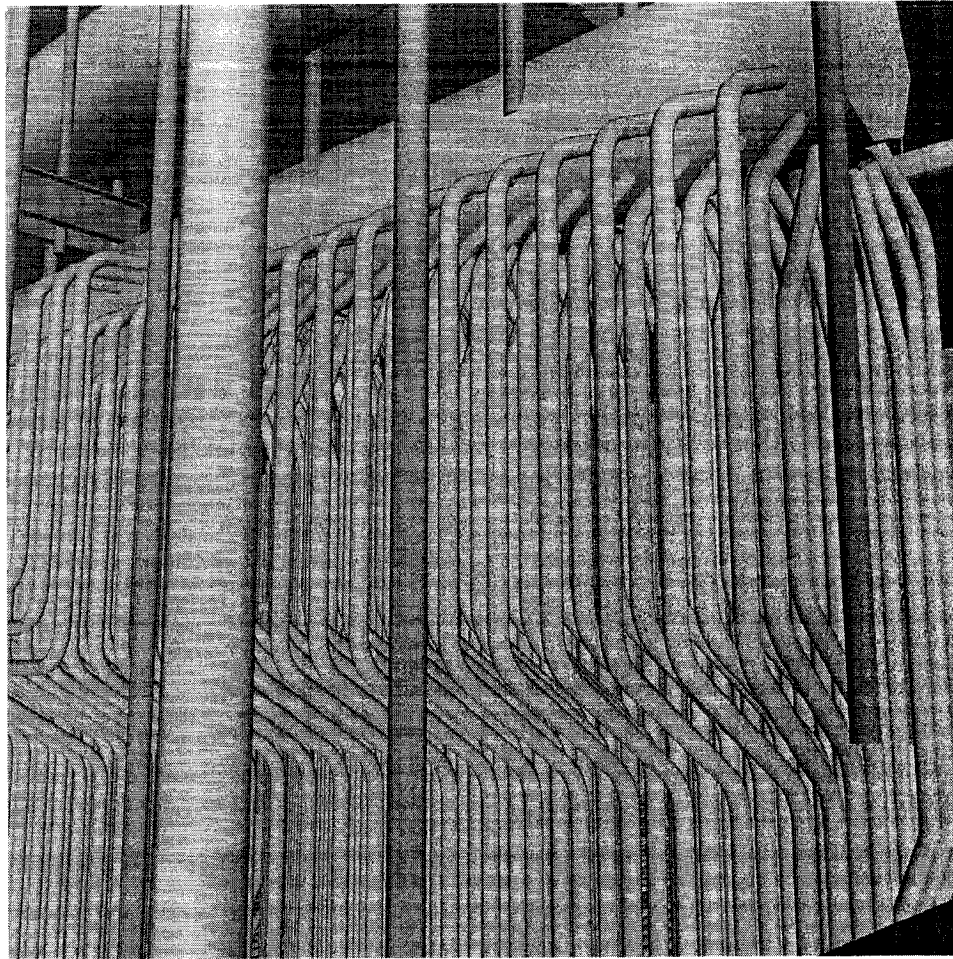


Figure 1.2: *Complex piping in the Power Plant model: This figure shows the dense piping structures present in the furnace room of the Power Plant model. Observe that each pipe occludes a very small portion of the furnace room, whereas a group of spatially proximate pipes can occlude most portions of the scene. The scene is composed of 13 million polygons.*

on environments composed of millions of polygons and high depth complexity. In terms of collision detection, we would like to compute interferences among objects composed of hundreds of thousands of polygons.

3. **General Environments:** We do not make any assumptions about the geometric representation of models. These 3-D models may have complex piping structures and may not contain large occluders; examples of such models are shown in Figs. 1.2 and 1.3. Further, it may be difficult to decompose these models into cells and portals. We also aim at designing collision detection algorithms capable of

handling polygon sets (or soups).

4. **Quality and Accuracy:** It is important to guarantee the final output - image quality in the case of rendering algorithms and accuracy of collision computations for simulation algorithms. Our goal is to develop conservative algorithms providing bounds on the quality or accuracy of the computations.
5. **Commodity Hardware:** Many interactive applications such as games target common users with access to commodity hardware. For the sake of practical applications, we aim at developing algorithms on commodity processors rather than on special-purpose hardware or supercomputers.

1.4 Prior Work and Challenges

Many efficient algorithms attempt to reduce the number of expensive operations on geometric primitives for achieving interactivity. Rendering algorithms attempt to reduce the number of geometric primitives sent to the graphics processors during each frame. Collision detection algorithms attempt to reduce the number of pair-wise interference tests between geometric primitives during each instance. We now give a brief overview of visibility-based computations for reducing the number of expensive operations in the above applications.

1.4.1 Visibility for Rendering Algorithms

Visibility computations have been well studied in computer graphics. Computing exact visible sets of geometric primitives is often expensive in complex environments (El-Sana et al., 2001). Alternatively, several culling strategies have been developed. The commonly used techniques include:

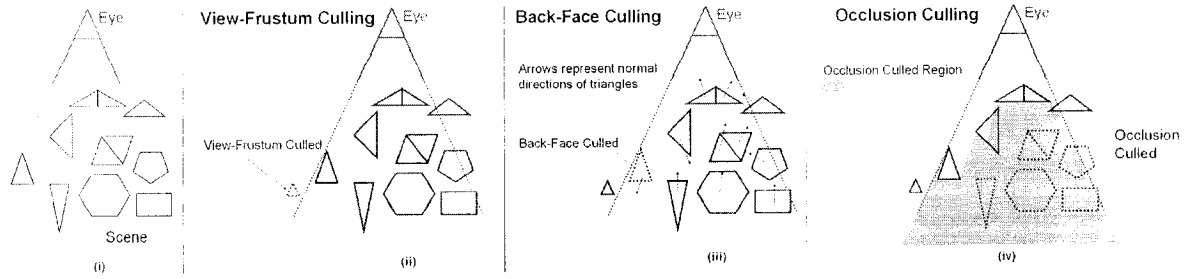


Figure 1.3: *Different culling techniques on a scene shown in (i): view-frustum culling in (ii), back-face culling in (iii), and occlusion culling in (iv).*

- **View-frustum culling** rejects primitives outside the view frustum. It is simple and fast, and the culling efficiency depends upon the number of geometric primitives within the view frustum. In many walkthrough applications, a large portion of the scene can be within the view frustum. In such scenarios, view frustum culling may not be effective.
- **Back-face culling** rejects primitives with normals facing away from the user. This technique requires geometric primitives with valid normals. In many walkthrough and virtual-reality applications, the user may be interested in viewing both sides of the geometric primitives, and this limits the use of back-face culling in these applications. In certain applications like shadow computations, however, it is possible to compute valid normals for geometric primitives dynamically, based on eye and light positions, and back-face culling can be applied.
- **Occlusion culling** rejects geometric primitives that are occluded by other geometric primitives. It is more complex and expensive compared to back-face culling and view-frustum culling (El-Sana et al., 2001) because visibility events are difficult to comprehend for even small changes in view position. It is primarily used in walkthroughs of large data sets with high depth complexity.

The application of these techniques to a simple scene is illustrated in Fig. 1.3.

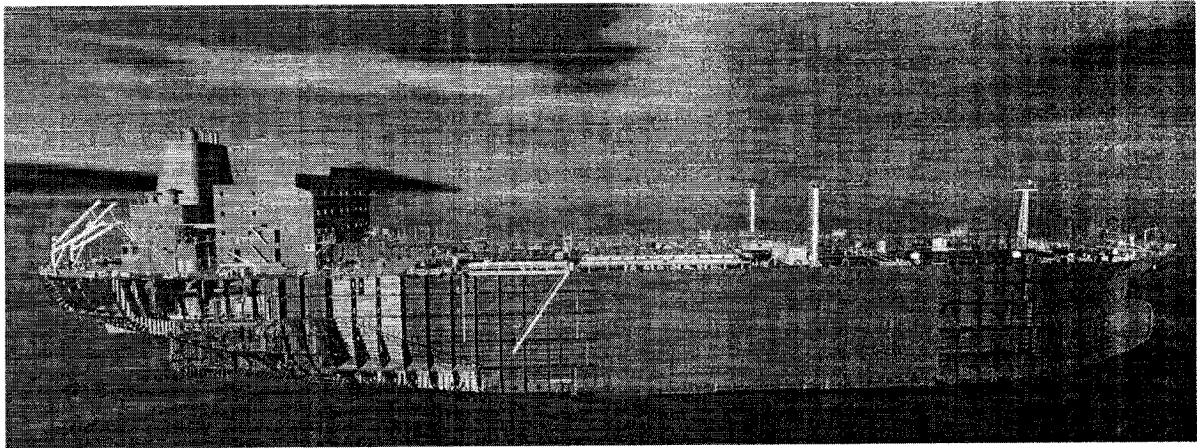


Figure 1.4: *Double Eagle Tanker model: This model is composed of 82 million polygons and exhibits high visual and depth complexity.*

Challenges and Issues: We are interested in rendering general and complex environments with high depth complexity. Moreover, these models may exhibit high visual complexity and consist of millions of polygons. An example is shown in Fig. 1.4. Further, the complexity of these models is increasing due to advances in acquisition, modeling and simulation technologies, and rendering these models at interactive rates is becoming increasingly difficult. Back-face culling and view-frustum culling techniques may not be effective due to the reasons mentioned above, and visibility culling techniques need to be designed for rendering these data sets at interactive rates.

Many visibility culling techniques have been developed for specialized environments. Examples include architectural and urban environments and environments with large convex occluders. These algorithms may not work well on general environments (Cohen-Or et al., 2001a) such as a Power Plant model consisting of many long and skinny pipes.

As mentioned in Section 1.2, object-space culling algorithms may not work well on general environments. On the other hand, image-space culling algorithms provide occluder fusion and, in practice, are well suited for general environments. They are mainly implemented on graphics processors, as GPUs are optimized for fast rendering.

Current image-space culling algorithms such as (Greene et al., 1993; Zhang et al., 1997b; Baxter et al., 2002) involve the following issues:

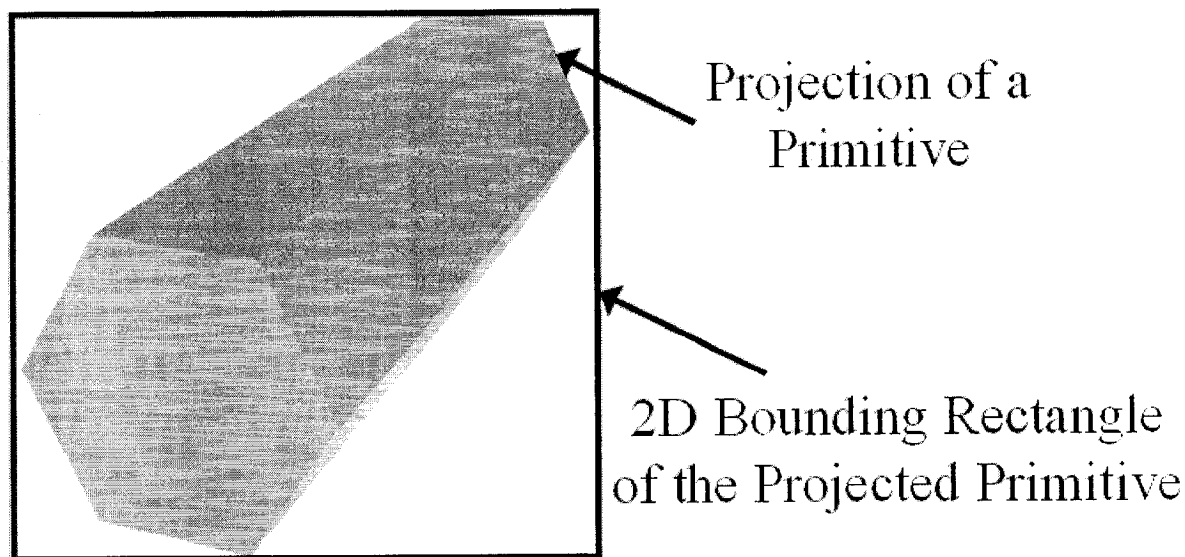


Figure 1.5: *This figure shows a simple primitive projected on the screen and its screen-space bounding rectangle. Observe that the screen-space bounding rectangle occupies a much larger screen space in comparison to the projected primitive and can be quite conservative for performing visibility computations.*

- **Frame-buffer Readbacks:** These can be expensive and are mainly bandwidth limited. They may involve graphics pipeline stalls and thereby, affect the throughput. These issues are discussed in a greater detail in Section 1.5.
- **Conservativeness:** Current image-space algorithms (Greene et al., 1993; Zhang et al., 1997b; Baxter et al., 2002) read back the contents of the frame-buffer and perform conservative occlusion culling by testing the 2-D bounding rectangles of occludee-representation projections (e.g., projections of bounding boxes of occludees) for visibility. 2-D bounding rectangles of projected primitives are often more conservative than projections of primitives as shown in Fig. 1.5.

1.4.2 Visibility for Shadow Algorithms

Shadows due to a point light source refer to portions of the scene that are not visible from the light source. Shadows add realism to a scene and are important in many interactive applications such as virtual walkthroughs and games. As these applications have high performance demands, recent research has focused on designing techniques for interactive shadow generation (Govindaraju et al., 2003a; Lloyd et al., 2004; Sen et al., 2003; Fernando et al., 2001; Stamminger and Drettakis, 2002). These techniques can be broadly classified into three categories:

- **Image-precision techniques** sample the scene at discrete positions and use the sampled representation for generating shadows. However, discrete sampling of the scene generates aliasing artifacts in many image-precision algorithms (e.g., shadow maps (Williams, 1978)).
- **Object-precision techniques** compute the exact shadow boundaries and thus avoid aliasing artifacts. Computation of shadow boundaries requires knowledge of shadow casters and shadow receivers in the scene (Govindaraju et al., 2003a; Lloyd et al., 2004). It may be difficult to compute these shadow casters and shadow receivers interactively in general environments. Further, computation of shadow boundaries can be expensive in complex environments.
- **Hybrid techniques** use a combination of object-precision and image-precision techniques for computing shadows (Govindaraju et al., 2003a; Sen et al., 2003).

Challenges and Issues: We are interested in generating high-fidelity, interactive shadows on general and complex models and using a few commodity PCs (typically three or fewer PCs). Although shadow generation is extensively studied, current techniques have not been well demonstrated in applications with the above requirements.

Image-precision techniques work well on general environments but suffer from aliasing artifacts. Object-precision techniques generate high-quality shadows, but achieving interactivity is challenging due to the large number of shadow casters and shadow receivers in a complex scene. There is a need to design shadow culling algorithms for computing shadow casters and shadow receivers that do not result in aliasing artifacts. Also, for generating shadows on large models, there is a need to investigate the integration of other rendering acceleration techniques such as level-of-detail-based algorithms (LODs) for shadow generation.

1.4.3 Visibility for Collision Detection

Collision detection is a fundamental problem in several interactive applications such as games, walkthroughs, etc. The problem of collision detection is very similar to the problem of visibility computations in an environment. Both problems are global by nature i.e., in the worst case, each primitive may be tested against all the remaining primitives. In other words, the visibility of a primitive is determined based on the location of other primitives. Similarly, the collision status of a primitive is determined based on the location of remaining primitives. Visibility computations have been applied for pruning non-interfering primitives (Hudson et al., 1997b; Gottschalk et al., 1996b; Rossignac et al., 1992). These approaches work well for rigid environments or environments consisting of closed objects.

Challenges and Issues: Although many approaches have been proposed for fast computation of colliding primitives, collision detection remains a bottleneck in several real-time simulations. These simulations include non-rigid simulations as well as simulations where objects change topology (e.g., breaking objects).

Object-space algorithms involve pre-processing and are designed to work well for rigid models (Hubbard, 1993; Quinlan, 1994; Beckmann et al., 1990; Ponamgi et al.,

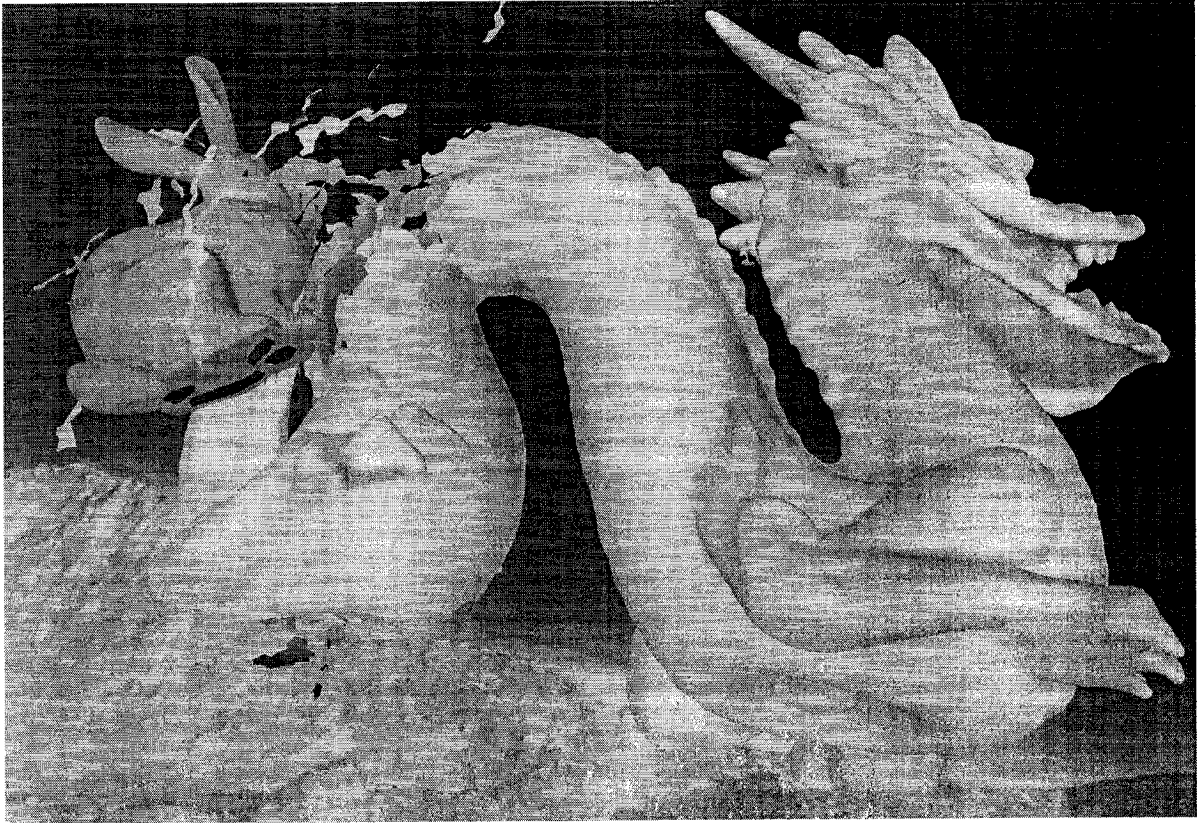


Figure 1.6: *This figure shows a snapshot of a dynamic simulation consisting of many objects in close proximity. These close-proximity scenarios are challenging for most interactive collision detection algorithms. In this simulation, as time progresses, there are more than a thousand objects in close proximity. These objects in the scene together are composed of 150K polygons.*

1997; Gottschalk et al., 1996a; Barequet et al., 1996; Held et al., 1996; Klosowski et al., 1998). Image-based algorithms (Baciu et al., 1998; Baciu and Wong, 2002; Heidelberger et al., 2003; Hoff et al., 2001; Knott and Pai, 2003; Myszkowski et al., 1995; Rossignac et al., 1992) (Shinya and Forgue, 1991; Vassilev et al., 2001) are well suited for non-rigid simulations. Most of the current image-based algorithms, however, either involve frame-buffer readbacks or are limited to closed models (Govindaraju et al., 2003b). In addition, the accuracy of most of these algorithms is limited by the viewport resolution or frame-buffer precision.

Prior algorithms utilizing visibility computations for collision detection involve simple heuristics and are quite conservative (e.g., view-frustum culling, shadow frusta, shadow volumes for collision pruning) in complex non-rigid simulations, where objects

may be in close proximity. An example is shown in Fig. 1.6. Hence, there is a need to design less conservative visibility-based collision detection algorithms that are reliable and work well on non-rigid models.

1.5 Graphics Processors

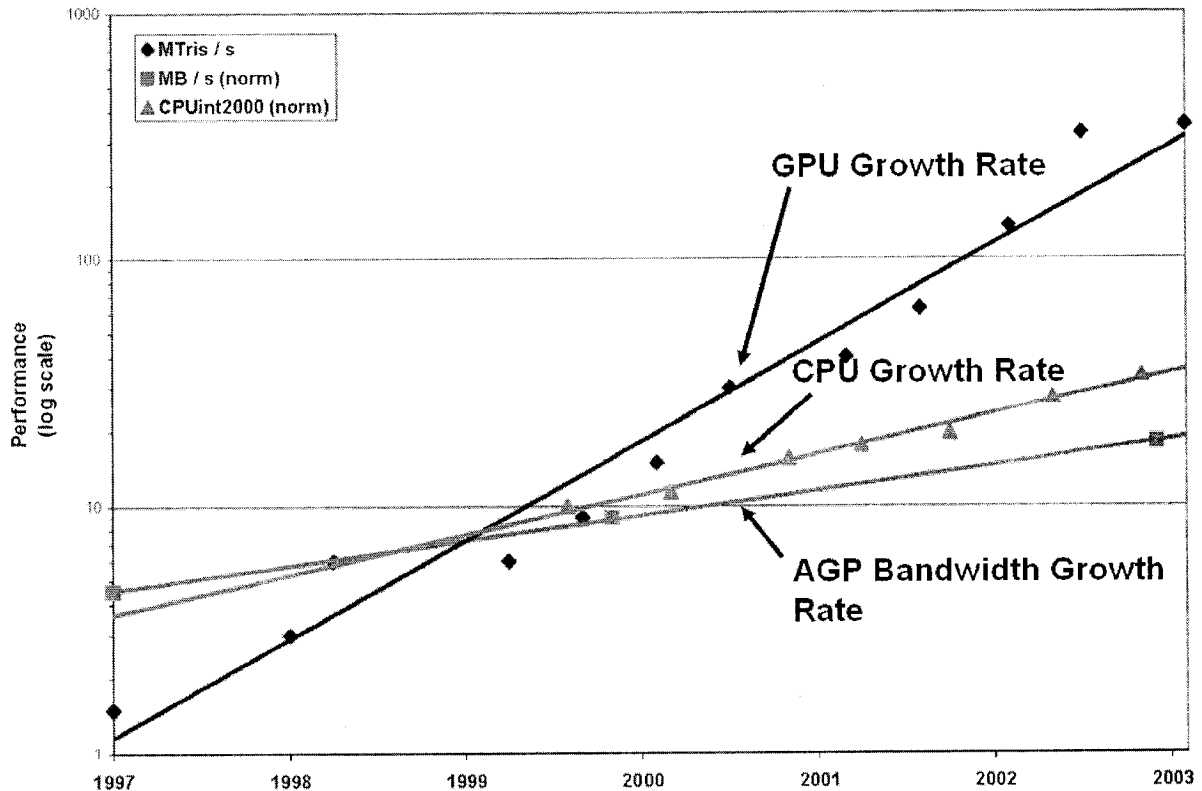


Figure 1.7: *Super Moore's law* : This figure shows the performance growth rates of GPU, CPU, and AGP bandwidth between GPU and CPU. GPU's computational power has been progressing at a rate more than Moore's law for CPUs. Courtesy Anselmo Lastra, UNC

One of our goals is to use commodity processors for performing visibility computations and solving the three problems. Graphics processing units (GPUs) are highly optimized for performing visibility computations from a viewpoint. These computations are typically performed on GPUs by rendering polygons. Current commodity graphics processors such as the NVIDIA GeForceFX 6800 Ultra can render several hundreds of millions of polygons per second. Moreover, their computational power seems to be progressing consistently at the rate of 2.5 - 3.0 times a year, much more than Moore's

law for CPUs. Fig. 1.7 shows the performance growth curve of GPUs, CPUs, and the bandwidth between GPU and CPU. In order to achieve these high-performance benchmarks, GPUs are designed with special-purpose hardware. In this section, we outline the various features of current GPUs useful for performing visibility computations.

1. **Memory Bandwidth:** A GPU is designed to transform the geometric description of a scene rapidly into the pixels on the screen that constitute a final image. Pixels are stored on the graphics card in a *frame-buffer*. The frame-buffer can be conceptually divided into three buffers according to the different values stored at each pixel:

- **Color buffer:** stores the color components of each pixel in the frame-buffer. Color is typically divided into red, green, and blue channels with an alpha channel that is used for blending effects.
- **Depth buffer:** stores a depth value associated with each pixel. The depth is used to determine surface *visibility*.
- **Stencil buffer:** stores a stencil value for each pixel. It is called the stencil buffer because it is typically used for enabling/disabling writes to portions of the frame-buffer.

The frame-buffer has a high memory bandwidth, and is accessed by various portions of the graphics pipeline. For example, NVIDIA GeForceFX 6800 Ultra has a 256-bit advanced memory interface to a high-speed memory with a data rate of 1.1 GHz, thus providing a peak memory bandwidth of 35.2 GBps. In contrast, current high-end main memories on PCs such as dual channel 128-bit DDR2 400 RDRAM have a peak bandwidth of 6.4 GBps. This fast memory interface on GPUs helps in achieving high fill rates. For example, an NVIDIA GeForceFX 6800 Ultra has a peak fill rate of 6.4 billion texels per second.

GPUs also exploit spatial coherence obtained in polygon rasterization for reducing the memory bandwidth. Examples include multi-sampling modes of rendering. The color, depth, and stencil buffers are stored in a compressed format, allowing for efficient reads and writes. Effectively, compression improves the available memory bandwidth on GPUs considerably.

Visibility computations on GPUs typically involve comparison operations on the frame-buffer data (e.g., depth values of primitives are compared against the depth buffer). These operations usually require data reads from and writes to the frame-buffer. The high memory bandwidth allows several operations to be performed in parallel.

2. **Pipeline:** The transformation of geometric primitives (points, lines, triangles, etc.) to pixels is performed by the graphics pipeline, consisting of several functional units, each optimized for performing a specific operation. Fig. 1.8 shows the various stages involved in rendering a primitive.

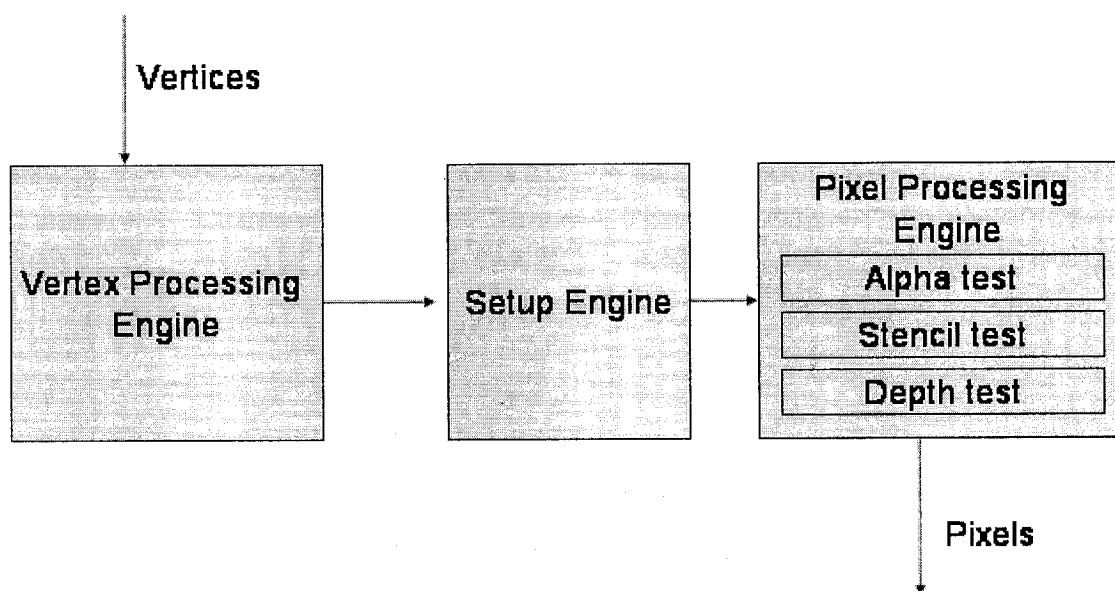


Figure 1.8: *Graphics architectural pipeline overview: This figure shows the various units of a modern GPU. Each unit is designed for performing a specific operation efficiently.*

- **Vertex Processing Engine:** This unit receives vertices as input and transforms them to points on the screen.
- **Setup Engine:** Transformed vertex data is streamed to the setup engine which generates slope and initial value information for color, depth, and other parameters associated with the primitive vertices. This information is used during rasterization to construct *fragments* at each pixel location covered by the primitive.
- **Pixel Processing Engines:** Before the fragments are written as pixels to the frame-buffer, they pass through the pixel processing engines or *fragment processors*. A series of tests can be used for discarding a fragment before it is written to the frame-buffer. Each test performs a comparison using a user-specified relational operator and discards the fragment if the test fails.
 - **Alpha test:** Compares a fragment’s alpha value to a user-specified reference value.
 - **Stencil test:** Compares the stencil value of a fragment’s corresponding pixel with a user-specified reference value.
 - **Depth test:** Compares the depth value of a fragment to the depth value of the corresponding pixel in the frame-buffer.

The relational operator can be any of the following: $=$, $<$, $>$, \leq , \geq , and \neq . In addition, there are two operators that do not require a reference value: *never* and *always*.

These comparison tests mainly determine the visibility status of a primitive based on our definition in Section 1.1. Aided by the power of comparisons, visibility could be used as an effective tool in solving many general-purpose problems such as database queries (Govindaraju et al., 2004), order statistics, mining of frequent items, etc. Furthermore, each stage of the graphics pipeline operates on a stream of

commands and data, improving the overall throughput of the system. Therefore, a GPU's architecture may be well suited for different streaming applications such as video processing, mining data streams, etc.

3. **Multiple Programmable Processors:** Current generations of GPUs have a vertex processing engine and pixel processing engine that are programmable. The user can supply a custom *vertex* or *fragment program* to be executed on each incoming input to these engines. For example, a fragment program can compute the alpha value of a fragment as a complex function of the fragment's other color components or its depth. These programmable processors have become powerful computational engines for many general-purpose computations such as linear algebra operations (Kruger and Westermann, 2003), sparse matrix solvers for conjugate gradient and multigrid methods (Bolz et al., 2003), etc.

Furthermore, current GPUs have multiple vertex and pixel processing engines. For example, an NVIDIA GeForce FX 6800 Ultra has 6 vertex processing engines and 16 pixel processing engines, each of which operates at 400 MHz. These powerful SIMD engines can process up to 600 million vertices per second and 6.4 billion texels per clock. In addition, some GPUs can perform twice the number of operations in certain modes, improving their rendering throughput. For example, an NVIDIA GeForce FX 6800 Ultra can process upto 32 pixels per clock in Z/Stencil-only modes. These optimizations can improve the rendering performance by nearly a factor of 2 in many visibility-based computations which typically perform Z/Stencil-only operations.

4. **Miscellaneous Features:** Graphics processors have special hardware features, some for improving the rendering performance, some for streaming computations, and some for adding rendering effects.

- **Texture Units:** Each pixel processing engine has access to texture units.

These units support a variety of operations on the color component of primitives such as random lookup, optional filtering etc. These operations are useful for generating rendering effects such as transparency, etc.

- **Caches:** Caches are used to save the bandwidth required for accesses to and from the texture units. Also, GPUs save transformation costs in vertex engines using pre-transform and post-transform vertex caches.
- **Early Z-Cull Hardware:** This feature is used early in the pipeline for terminating fragments not contributing to the final output. Examples of these technologies include the HyperZ III from ATI and early Z-cull from NVIDIA. This specialized hardware is integrated into the graphics pipeline for improving the rendering throughput.
- **Shadow Computations:** These features are extensively used in gaming and other interactive applications. Many vendors provide hardware support for algorithms such as shadow maps and are adding new features such as NVIDIA's Ultra-shadow technology (D03 depthbounds, 2003). Many visibility-based algorithms such as CSG rendering (Guha et al., 2003), range queries in databases (Govindaraju et al., 2004), etc., use these features extensively.
- **Occlusion Queries:** These queries return the number of fragments that are not discarded in the graphics pipeline (NVocclusion Query, 2001). As these queries return an integer, they have low bandwidth requirements. Recent hardware implementations allow asynchronous issue of multiple occlusion queries at a time and read back the results at a later time. Asynchronous processing reduces the pipeline stalls, improving the overall performance. Many applications disable the depth writes and determine the visibility status of a primitive using an occlusion query. These applications need to perform only data reads from frame-buffer memory and therefore, are faster than those performing depth or color writes.

These hardware functionalities of the GPU enable them to perform fast visibility computations. Increasingly, their potential is being realized in solving many general-purpose problems. GPUs, however, suffer from some limitations:

- **Precision:** Current GPUs support floating point textures with 24-bit or 32-bit precision. Many algorithms perform computations on input data stored in these textures and store the output results in textures. The precision of these algorithms is limited to 24-bit or 32-bit floating point numbers. However, many scientific applications require double precision for arithmetic calculations.

GPUs are also used for performing geometric queries such as proximity computations. These algorithms rasterize the input geometric primitives and perform computations on the sampled screen-space representations. The accuracy of these algorithms is limited by the precision of viewport (which is limited to $4K \times 4K$ on current GPUs like NVIDIA GeForce FX 6800 Ultra, thus accounting for 12 bits) and depth buffer (which is limited to 24-bit fixed precision on current GPUs).

- **Conditionals:** The pixel and vertex processors are SIMD engines. Therefore, evaluation of conditionals in fragment and vertex processors can be inefficient due to lock-step computation (Foley et al., 1990).
- **Data Rearrangement:** Current fragment processors do not support arbitrary writes, and therefore data rearrangements are expensive on GPUs.
- **Frame-Buffer Readbacks:** These can be expensive on current systems (Govindaraju et al., 2003b; Knott and Pai, 2003). Frame-buffer readbacks are mainly bandwidth limited. With the advent of PCI-Express, frame-buffer readbacks may become fast. However, frame-buffer readbacks involve graphics pipeline stalls and therefore affect the throughput. Also, the bandwidth growth rate is much lower than Moore's law for CPUs (as shown in Fig. 1.7), and since the performance of al-

gorithms involving frame-buffer readbacks is bandwidth limited, their performance improves at a rate lower than Moore’s law.

1.6 Thesis Statement

Efficient algorithms can be designed to perform visibility computations in complex environments and used for interactive walkthroughs, shadow generation, and collision detection on current commodity graphics hardware.

1.7 New Results

In this section, we highlight our key results. Our results in solving each of the three problems are summarized below.

Interactive Walkthroughs

1. **Interactive Performance on Complex Environments:** We present a parallel occlusion culling algorithm on three commodity GPUs for interactive walkthroughs of complex environments. In order to achieve high frame rates, our algorithm exploits frame-to-frame coherence to perform occluder selection, to traverse scene hierarchy, and to reduce communication bandwidth between the three PCs. Our algorithm achieves interactive performance on three complex environments: a Power Plant model with 13 million triangles, the Double Eagle Oil Tanker model consisting of 82 million triangles and a portion of Boeing 777 consisting of 20 million triangles.
2. **General Environments:** Our algorithm is image-based and therefore works well on general environments (as discussed in Section 1.2). As compared to earlier image-based algorithms, our approach involves no frame-buffer readbacks. As a

result, our algorithm does not involve large pipeline stalls in comparison to stalls due to frame-buffer readbacks. Moreover, our algorithm is less conservative than prior image-based occlusion culling algorithms.

3. **Quality :** Our algorithm integrates with LODs and is able to render complex models at interactive frame rates while sacrificing little image quality. The image quality is computed as the deviation of projection of simplified primitives from the projection of original primitives and is measured in terms of pixels of error. In practice, we are able to render complex models using a few pixels of error at high resolutions (typically 5 or fewer pixels of error at 1000×1000 screen resolution).
4. **Commodity Hardware:** Our algorithm works on three commodity GPUs. Its performance mainly depends upon the performance of occlusion queries and rasterization rate, both of which have been improving at a rate greater than Moore’s law. Therefore, in the future, with improvements in GPU performance, we may be able to handle more complex models.

Shadow Generation

1. **Interactive Performance on Complex Environments:** We present a parallel shadow culling algorithm on three commodity GPUs for interactive shadow generation in complex environments. We use our visibility culling algorithms designed for interactive walkthroughs to compute potential shadow casters and shadow receivers in the scene. In order to achieve high frame rates, we improve our visibility culling algorithm to compute less conservative sets of shadow casters and shadow receivers. We also present a novel shadow culling algorithm which eliminates shadow casters and shadow receivers that do not contribute to the shadow boundaries. Moreover, we present a parallel pipelined architecture using three GPUs for fast shadow generation. We demonstrate the performance of our algorithm

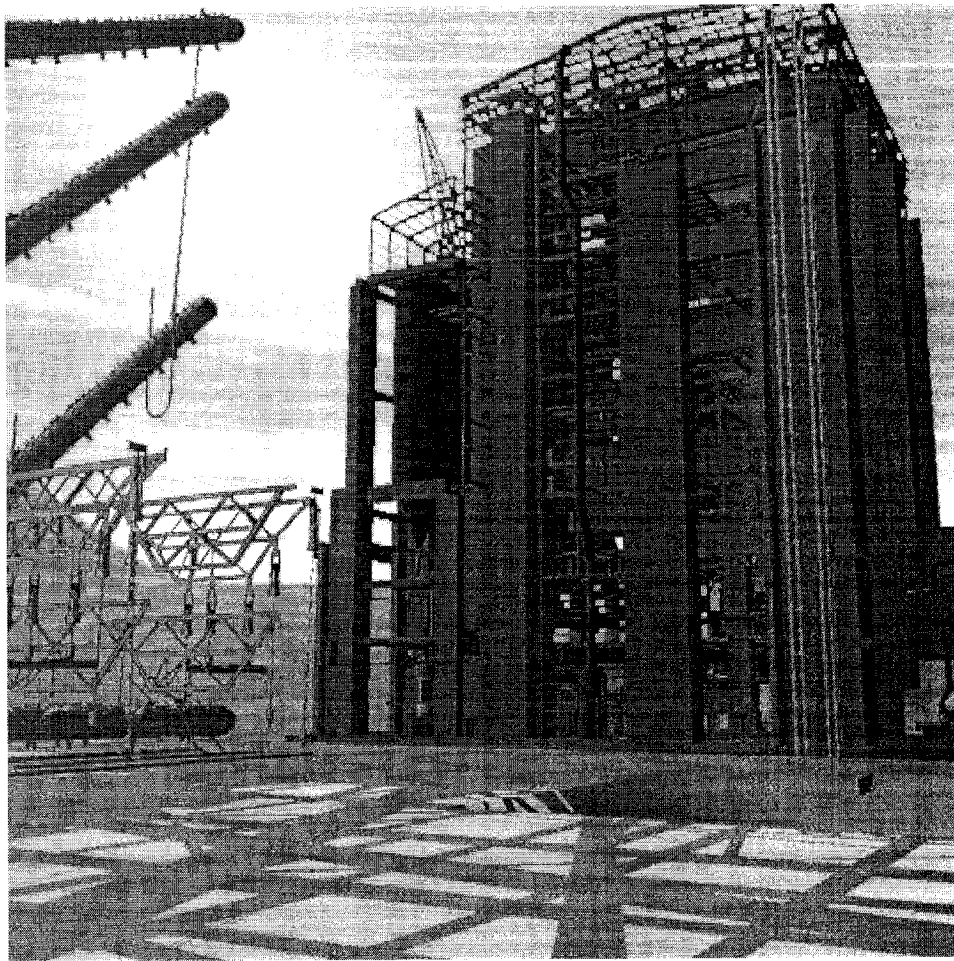


Figure 1.9: A snapshot generated from the application of our hybrid shadow-generation algorithm to the Power Plant model (12.7 million triangles). Our interactive algorithm is able to generate shadows at an average frame rate of 10 fps using three PCs, each with an NVIDIA GeForce 4 graphics card.

on three complex models: a house model with a million triangles, a Power Plant model with 13 million triangles and a Double Eagle Oil Tanker model consisting of 82 million triangles. Fig. 1.7 demonstrates the performance of our algorithm on the Power Plant model. In practice, we are able to render these models at 7 - 10 frames per second using three NVIDIA GeForce 4 GPUs.

2. **General Environments:** Our shadow culling algorithm is image-based and works well on general environments. Our approach involves no frame-buffer read-backs and hence is not read back bandwidth-limited.

3. **Integration with LODs:** We have integrated LODs with our visibility culling algorithms. In practice, we are able to achieve interactive frame rates with little loss in image quality (typically, 0 - 20 pixels of error at 512×512 screen resolution).
4. **Commodity Hardware:** Our algorithm uses three commodity GPUs. The performance of our shadow culling algorithms depends mainly on the performance of occlusion queries and rasterization rate.

Collision Detection

1. **Interactive Performance on Complex Environments:** We present a simple and efficient collision detection algorithm that uses visibility computations on GPUs for performing fast collision detection on complex non-rigid models. Our algorithm uses a novel two-pass rendering algorithm for pruning objects that are not in close proximity. We have demonstrated the performance of our algorithm on complex environments consisting of multiple moving objects and objects undergoing non-rigid motion. These environments consist of tens of thousands of polygons and may contain objects in close proximity. In practice, we are able to compute all collisions in a few milliseconds (typically 3 - 50 ms).
2. **General Environments:** Our algorithm is capable of handling all simulations, non-rigid as well as rigid. We do not make any assumptions on input geometric primitives and can handle triangulated models that are convex or non-convex, open or closed, as well as objects that have dynamically changing topologies. Furthermore, our algorithm can compute self-intersections in complex non-rigid simulations.
3. **Quality and Accuracy:** We present a hybrid algorithm computing a set of potentially colliding primitives using an image-precision algorithm on GPU and computing pairs of colliding primitives using an object-precision algorithm on CPU.

Our image-based collision detection algorithm is reliable and *does not miss* a collision up to floating point precision.

4. **Commodity Hardware:** Our algorithm uses visibility computations on commodity graphics hardware. Further, our algorithm involves no frame-buffer read-backs, and its performance depends mainly on the performance of occlusion queries and rasterization rate.

1.8 Organization

The rest of the thesis is organized in the following manner:

- **Chapter 2** surveys the previous work in the areas of large-model rendering, shadow generation and collision detection.
- **Chapter 3** describes our algorithm *Occlusion switch* for performing interactive walkthroughs of complex environments.
- **Chapter 4** presents our visibility-based algorithms for generating high-fidelity hard shadows on complex models at interactive frame rates using GPUs.
- **Chapter 5** presents our visibility-based algorithms for performing fast collision computations in complex simulations.
- **Chapter 6** concludes with the major results of the thesis and discusses problem areas for future research.

Chapter 2

Related Work

Visibility from a point is well studied in computer graphics, vision, computational geometry and related areas. It is beyond the scope of this thesis to survey the vast literature exhaustively. We focus primarily on recent techniques designed to work well for large-model rendering, shadow generation and collision detection. As much has been written on these three topics, we briefly outline the details of the related algorithms. We also highlight the advantages and disadvantages of these algorithms. More extensive surveys are available for the interested reader (Bittner and Wonka, 2003; Cohen-Or et al., 2001a; Hasenfratz et al., 2003; Lin and Gottschalk, 1998; Jimenez et al., 2001; Lin and Manocha, 2003; Cohen and Manocha, 2003).

2.1 Large-Model Rendering

Occlusion culling is needed for interactive display of scenes with high depth complexity. Many visibility culling algorithms have been proposed to render scenes with high polygon count and depth complexity. These algorithms are used primarily for reducing the load on a graphics pipeline by culling polygons that are not visible to the user. Alternatively, several other techniques have been proposed for reducing the load on a graphics pipeline:

- rendering low polygon count approximations.
- approximating geometric primitives using multiple image-based representations,
- distributing the rendering load on multiple graphics pipelines or multiple processors.

In this section, we briefly survey the previous work in these areas.

2.1.1 Polygonal Simplification

Polygonal simplification techniques aim at reducing the geometric complexity of a polygonal model to a level that can be rendered at interactive frame-rates. Each object is represented using several levels-of-detail varying from coarse approximations, which drastically simplify the object, to fine approximations, which closely represent the original object. Depending on the level-of-detail, an approximation may consist of low or high polygon count.

Many algorithms have been proposed to perform polygonal simplification. These algorithms can be classified based on different properties: preservation of topology or not, different error-metrics used in performing simplification, etc.

Most of these algorithms perform decimation operations for computing different levels-of-detail. These operations include vertex removal, vertex clustering, vertex merging, edge collapses and face collapses (Cohen et al., 1996; Garland and Heckbert, 1997; Hoppe, 1996; Rossignac and Borrel, 1993).

Cohen et al. (Cohen et al., 1996) compute an outer envelope and an inner envelope bounding the surface mesh, and simplification is performed by progressively removing the vertices. After each vertex removal, the mesh is re-triangulated. A vertex removal is valid if the re-triangulated mesh does not intersect the envelopes.

Hoppe et al. (Hoppe, 1996) generate progressive meshes by performing a sequence of edge-collapse operations. Each edge-collapse operation replaces an edge with a ver-

tex, and re-triangulates the mesh by replacing the edge vertices with the new vertex. Progressive meshes encode a sequence of edge-collapse operations. Given a progressive mesh and a sequence of edge-collapse operations, the mesh can be refined progressively.

Each decimation operation re-triangulates the resulting hole and is associated with an error. The quality of the final image depends upon the error associated with the simplified representations. Cohen et al. (Cohen et al., 1997) describe simplification techniques for computing error associated with surface attributes (e.g., textures) in addition to the geometric error associated with simplified representations. Garland and Heckbert (Garland and Heckbert, 1997) present an efficient simplification algorithm based on error quadrics. Error quadrics are 4×4 matrices that compute the distance from a point to one or more planes that define the quadric. The simplification algorithm initially computes an error quadric for each polygon in the mesh. Simplified representations are computed by performing edge-collapse operations. Each edge-collapse operation replaces an edge with a new vertex and re-triangulates the mesh. The error associated with this edge-collapse is computed as the distance from the new vertex to each of the affected polygons in re-triangulation. The error associated with the edge-collapse operation is computed for each edge in the object. Then, the algorithm proceeds in a greedy manner by selecting the edge-collapse operation that generates lowest error.

The simplification process is performed as a pre-process. Traditionally, for each object, multiple simplified representations are created. These are often referred to as static LODs. At run-time, the system selects the LOD that satisfies the user-defined criteria such as distance from the viewer.

The use of static LODs can lead to many problems:

- **Large Objects:** If the object is large with respect to the view-point, it is difficult to represent the object using a single level-of-detail as portions of an object that are closer to the view-point require higher detail than portions of an object that are farther away from the viewer.

- **Multiple Small Objects:** Representing each object using a simplified representation may not reduce the polygon count drastically. For example, many objects that are in close proximity and far away from the viewer could be represented using a single representation, and thereby obtaining higher degrees of simplification as opposed to simplifying each object individually.
- **Popping Artifacts:** Each object represents a discrete level-of-detail and as the user moves, the LOD representation may switch to coarser or finer LODs. Switching the LOD may result in undesirable popping artifacts.

Some of these issues can be resolved using view-dependent simplification (VDS) techniques (Luebke and Erikson, 1997; Hoppe, 1997; Xia and Varshney, 1996). Instead of generating LODs as a pre-process, a view-dependent tree which encodes the order of decimation operations is created and used at run-time for choosing the appropriate level-of-detail. Using this approach, a fine-grain fidelity control is provided to the user or the rendering application. Given a fixed polygon budget, the VDS algorithm simplifies portions of objects close to the viewer less drastically and simplifies portions further away more drastically. A VDS algorithm also generates view-dependent approximations with smaller polygon count by merging several objects together. There are, however, a few disadvantages of using VDS in comparison to static LODs. The computation of dynamic LODs at run-time requires higher CPU resources than the static LODs. Moreover, it is more difficult to incorporate rendering-acceleration techniques such as vertex arrays along with VDS than with static LODs.

Few algorithms have been proposed that perform geometric simplification by computing a hierarchy of levels-of-detail (Erikson and Manocha, 1998; Clark, 1976; Hoppe, 1998). These algorithms inherit some of the advantages of VDS and static LODs. A scene hierarchy is constructed and represented using a scene graph. Each intermediate node is simplified and represented using a sequence of LODs. These are referred to as hierarchical LODs (HLODs). (Erikson and Manocha, 1998) use error quadrics (Garland

and Heckbert, 1997) to perform simplification. At run-time, the scene graph is traversed top-down, and nodes that satisfy the criteria (e.g., error threshold) are selected. The selected nodes are rendered. Each HLOD can be considered a discretized node in the view-dependent tree. HLODs inherit the advantages of static LODs. In addition, multiple small objects are simplified to generate HLODs. At higher error thresholds, however, switching between HLODs might introduce popping artifacts.

Although LOD algorithms render simplified representations of objects and thus reduce the polygon count, in complex environments with high depth complexity, the selected simplified representations may still consist of a large polygon count. Moreover, many of these polygons may not be visible to the user. Therefore there is a need to integrate visibility computations with simplification techniques to achieve interactive frame rates on these large data sets.

2.1.2 Image-Based Representations

Image-based rendering (IBR) algorithms render simple, sampled approximations for groups of primitives. Often these sampled approximations are represented using images with or without per-pixel depth. In complex environments consisting of millions of polygons, processing pixels in a sampled image may be faster than drawing the original geometric primitives.

Most IBR techniques are based on the plenoptic function formulation (Adelson and Bergen, 1991). Plenoptic functions are used to describe light transport in a scene. The plenoptic function P computes the intensity value p at a point (x, y, z) as $p = P(x, y, z, \theta, \phi, \lambda, t)$ where (θ, ϕ) defines the view direction V , λ is the wavelength and t is the time. Most IBR algorithms assume that the environment is static and pre-process the scene to generate many image-based representations sampled at different locations and view positions. Given a view direction V_{new} and the plenoptic functions at nearby sample locations along view directions $V_0, V_1, V_2, \dots, V_N$, the goal of these algorithms is

to construct an approximation for the plenoptic function at V_{new} . Many techniques have been proposed based on this notion. A few common image-based rendering approaches include texture mapping, image warping (Aliaga and Lastra, 1999; Max and Ohsaki, 1995; McMillan and Bishop, 1995), layered depth images (Shade et al., 1998), light fields (Levoy and Hanrahan, 1996; Buehler et al., 2001), billboards (Decoret et al., 1999), and texture depth meshes (Aliaga et al., 1999; Darsa et al., 1998; Jeschke and Wimmer, 2002; Sillion et al., 1997; Wilson et al., 2001). These techniques replace distant geometry by using image-based representations, and different image-based representations have been proposed. Some of these representations include point primitives, layered depth images (LDIs) and textured depth meshes (TDMs). We briefly outline some of these techniques whose goal is to render complex models at an interactive frame rate.

Point-primitive-based techniques are designed primarily for rendering highly sampled primitives (for, e.g., range-scanned data). These techniques mainly use points for rendering. Recent large-model rendering techniques based on point primitives include surfels (Pfister et al., 2000) and QSplat (Rusinkiewicz and Levoy, 2000). Pfister et al. (Pfister et al., 2000) propose simple surface elements called surfels for rendering geometric primitives. Surfels are points sampled on the surfaces of complex geometric models. The samples are computed along three orthographic views in a pre-process. While pre-processing the data sets, computation-intensive tasks such as texture, bump, or displacement mapping are performed. Therefore rendering costs based on texturing are eliminated at run-time. Surfels work well on models with high surface detail and may not work well on flat surfaces.

Some techniques have been proposed for rendering massive models using TDMs (Aliaga et al., 1999; Wilson and Manocha, 2003). TDMs are simplified meshes created from the sampled depth values in a depth image. A TDM is a height field with respect to the sample view position. The MMR system (Aliaga et al., 1999) is a massive model rendering system that integrates multiple rendering-acceleration techniques, including

visibility culling, geometric levels-of-detail, and TDMs. The system performs extensive pre-processing and partitions the model into view-point cells. Distant geometry is approximated by TDMs generated from the center of view-point cell. However, as the viewer moves from one cell to another cell, some visibility artifacts can occur. Wilson and Manocha (Wilson and Manocha, 2003) propose an incremental algorithm for computing image-based simplifications of large environments. The scene is sampled based on visibility, and TDMs are constructed at each view-point placed at the center of the cell. Further, an optimization function is proposed for selecting the sample points and is used for minimizing artifacts such as skins and cracks in reconstruction. A primary advantage of TDMs is that they are compatible with the standard rendering pipeline. Due to discrete sampling, however, algorithms using TDMs can result in some visibility artifacts.

2.1.3 Visibility Culling

Visibility culling techniques aim at rejecting geometric primitives not visible to the user. In the literature of computer graphics, there are three classical strategies for performing visibility culling (refer to Fig. 1.3):

- **View-frustum Culling** techniques aim at rejecting geometric primitives outside the view frustum. Many efficient hierarchical techniques have been proposed for performing fast view-frustum culling (Clark, 1976; Rusinkiewicz and Levoy, 2000; Greene et al., 1993). The amount of culling obtained using view-frustum culling is dependent on the number of geometric primitives within the user's view frustum.
- **Back-face Culling** techniques aim at rejecting geometric primitives whose normals face away from the user. The technique is limited to scenes consisting of geometric primitives with valid normals. On current commodity graphics processors, back-face culling is supported in hardware, and this improves its performance

considerably. Kumar et al. (Kumar et al., 1999) present an algorithm for performing hierarchical back-face computation.

- **Occlusion Culling** algorithms aim to cull away a subset of the primitives that are occluded by other primitives and therefore are not visible from the view-point. View-frustum culling and back-face culling depend on the location or normal of a geometric primitive. In contrast, occlusion culling depends upon the inter-relationship of a primitive with other geometric primitives and therefore is more complex than the other two techniques.

At a broad level, these visibility algorithms can be classified further as *point-based* algorithms (Greene et al., 1993; Zhang et al., 1997b; Baxter et al., 2002) or *region-based* algorithms (Durand et al., 2000; Schaufler et al., 2000; Wonka et al., 2000). Point-based algorithms reject geometric primitives that are not visible from a given point. Region-based algorithms reject geometric primitives that are not visible from any point in a given region. As it is expensive to perform visibility computations for an entire region, many of the region-based algorithms pre-compute the potentially visible sets for each region and render these sets at run-time for any view-point in the region. This imposes a further restriction that the input scene is appropriately decomposed into regions, and visibility computations can be performed efficiently for each region.

Each of these strategies has its own advantages and disadvantages. As region-based algorithms involve pre-computation, they may not work well on dynamic environments with moving objects in comparison to point-based algorithms. Also, region-based algorithms are more conservative than point-based algorithms. In addition, region-based algorithms can require long pre-processing and significant storage overhead. As region-based algorithms pre-compute visible geometric primitives, they are more suitable for out-of-core algorithms which require disk-to-memory pre-fetching of visible geometry.

In the literature of computer graphics and computational geometry, these visibility culling strategies have been studied extensively. In the remaining section, we focus

primarily on point-based occlusion culling algorithms and briefly discuss region-based occlusion culling algorithms. For detailed studies, refer to recent surveys (Cohen-Or et al., 2001a; Bittner and Wonka, 2003).

Point-Based Visibility Algorithms

Point-based algorithms have been designed for handling specialized environments or general environments.

- **Specialized Environments:** Many occlusion culling algorithms have been designed for specialized environments, including architectural models based on cells and portals (Airey et al., 1990; Teller, 1992) and urban data sets composed of large occluders (Coorg and Teller, 1997; Hudson et al., 1997a; Schaufler et al., 2000; Wonka et al., 2000; Wonka et al., 2001). These algorithms exploit the special characteristics exhibited by architectural environments and urban data sets. For example, many architectural scenes are naturally organized into rooms with doors and windows, and urban environments have large occluders. As these algorithms depend on the scene characteristics, they may not obtain significant culling on large environments composed of a number of small occluders.
- **General Environments:** Algorithms designed for these environments do not make any assumptions about the characteristics of the scene (Greene et al., 1993; Zhang et al., 1997a; Baxter et al., 2002; Govindaraju et al., 2003c).

Point-based algorithms perform visibility computations either in object space or image space or a combination of both techniques (hybrid). Next, we give a brief overview of these algorithms.

Object-Space Algorithms

These algorithms use geometric representations based on original primitives to perform culling. Examples of these representations include binary spatial partitioning (BSP)

trees, shadow frusta of occluders, etc. Some of the recent object-space algorithms for performing walkthroughs include cells and portals (Luebke and Georges, 1995), large convex occluders (Coorg and Teller, 1996; Coorg and Teller, 1997), shadow-frusta culling (Hudson et al., 1997a), and BSP-tree culling (Bittner et al., 1998).

Cells and portals are used mainly for rendering architectural scenes as these scenes can be decomposed easily into cells (rooms) connected by portals (windows or doors). Visibility computations use the observation that from any given cell, other cells are visible only via portals. Many earlier algorithms have been designed to pre-compute potentially visible set (PVS) for each cell (Airey et al., 1990; Teller, 1992). Luebke and Georges (Luebke and Georges, 1995) extend these ideas to compute PVS from a view-point at run-time. The algorithm proceeds by rendering the cell containing the view-point. The portals of the cell within the view frustum are identified. New cells and portals that are visible through these portals are clipped against their frusta. These newly clipped portals are added, and the algorithm proceeds in a recursive manner. The algorithm terminates when there are no visible portals.

Coorg and Teller (Coorg and Teller, 1996; Coorg and Teller, 1997) use a subset of large convex occluders to cull portions of the scene that are not visible to the user. The visibility status of occludees is determined by tracking a subset of visual events. Temporal coherence is used for incrementally computing these visual events. The algorithm assumes that the occluders are convex and large, and therefore it may not work well on large environments with many small objects.

Hudson et al. (Hudson et al., 1997a) propose a visibility culling algorithm by testing whether the occludee representations are enclosed within the shadow frusta of occluders. The occluder set is pre-computed using a spatial-partitioning algorithm. The scene is represented using a hierarchy, and each node contains the bounding box of the enclosed geometric primitives. At run-time, the approach chooses a set of occluders and computes their shadow frusta. The scene hierarchy is traversed top-down, testing whether the

bounding box of the node is enclosed within the shadow frustum of some occluder. (Bittner et al., 1998) improve the work of (Hudson et al., 1997a) by combining shadow frusta of occluders using BSP trees.

Image-Space Algorithms

Image-space algorithms cull away portions of the scene by testing them against discrete image representations. As portions of a scene are rendered, different regions of the image are filled and used for culling other portions of the scene. A primary advantage of using image-space algorithms is occluder fusion: the effect of obtaining the combined occlusion effect of multiple occluders. Therefore these algorithms are well suited for handling general environments that may consist of a large number of small objects. Recent image-based algorithms include ray casting, hierarchical Z-buffer (HZB) (Greene et al., 1993), hierarchical occlusion maps (HOM) (Zhang et al., 1997b), algorithms using multiple graphics pipelines (Baxter et al., 2002), and others based on image-based visibility queries (Bartz et al., 1999; Klosowski and Silva, 2001; Greene, 2001; Meissner et al., 2002; Hillesland et al., 2002).

Ray casting is a simple image synthesis algorithm that determines the first polygon in the scene intersecting the eye ray for each pixel. For each pixel, a ray originating from the eye and passing through a point in the pixel determines an eye ray. At run-time, the eye ray traverses the scene hierarchy, and polygons in the nodes of the hierarchy are tested for ray intersection. The algorithm can be highly parallelized, and spatial and temporal optimizations can be used, leading to well-optimized implementations (Parker et al., 1999; Foley et al., 1990; Wald et al., 2001; Purcell et al., 2002).

Z-buffer is a popular hidden surface removal algorithm that is supported in current graphics cards. The algorithm uses a discretized image representation and for each pixel, maintains the nearest depth value of all polygons that project onto the pixel. In modern graphics cards, an optimized architecture is designed to implement the algo-

rithm. (Greene et al., 1993) proposed a hierarchical implementation of the Z-buffer. An image-space Z pyramid is constructed, and each primitive is tested for visibility against the Z pyramid. The Z pyramid is constructed by successively applying a 2×2 filter recursively on the levels of the Z Pyramid beginning with the finest level. The filter computes the maximum value of the 4 adjacent values in a level. The original Z-buffer is used as the finest level of the pyramid. The Z-pyramid construction terminates when the filtering operation reaches the coarsest level, which contains one single value. For each polygon tested for visibility, the finest-level pixel sample of the pyramid containing the screen-space bounding rectangle of the polygon is computed. If the nearest Z value of the polygon is farther than the Z value of the pixel, the polygon is determined to be invisible. Otherwise, the algorithm recurses down the Z pyramid, testing the Z values of all pixels that cover the screen space bounding rectangle of the polygon. If the algorithm reaches the finest level and the polygon is not rejected, it is considered potentially visible. The algorithm has the advantage of occluder fusion and therefore is well suited for general environments. However, as the algorithm uses the screen-space bounding rectangle of a polygon and only its nearest Z value for testing its visibility, the rejection test can result in many false negatives, leading to large potential visible sets in complex environments (i.e., overly conservative).

(Zhang et al., 1997b) used hierarchical occlusion maps for performing visibility culling of general environments. An occlusion map stores the opacity value of a rectangular block in an image. A hierarchy of occlusion maps is constructed by filtering operations similar to (Greene et al., 1993). The filtering operation computes the average of the 2×2 blocks in each level. During each frame, a set of occluders is rendered, and a hierarchy of occlusion maps is constructed. These occlusion maps are used to cull portions of the scene not visible to the user.

(Baxter et al., 2002) use an additional graphics pipeline for performing visibility computations. Baxter et al. (Baxter et al., 2002) propose a two-pipeline-based occlu-

sion culling algorithm for interactive walkthrough of complex 3-D environments. The resulting system, GigaWalk, uses a variation of the two-pass HZB algorithm that reads back the depth buffer, and then computes the hierarchy in software. GigaWalk has been implemented on an SGI Reality Monster and uses two Infinite-Reality pipelines and three CPUs. Due to the high read back cost on current GPUs, a distributed implementation of the algorithm on commodity processors can be slow (Govindaraju et al., 2003c).

A number of image-space visibility queries have been added by manufacturers to their graphics systems to accelerate visibility computations. These include the HP occlusion culling extensions, item buffer techniques, ATI's HyperZ extensions, etc. (Bartz et al., 1999; Klosowski and Silva, 2001; Greene, 2001; Meissner et al., 2002; Hillesland et al., 2002). All these algorithms use the GPU to perform occlusion queries as well as render the visible geometry. As a result, only a fraction of a frame time is available for rasterizing the visible geometry, and it is non-trivial to divide the time between performing occlusion queries and rendering the visible primitives. If a scene has no occluded primitives, this approach will slow down the overall performance. Moreover, the effectiveness of these queries varies based on the model and the underlying hardware.

Hybrid Algorithms

These algorithms use a combination of object-space algorithms and image-space algorithms for performing visibility culling. Recent approaches include HZB (Greene et al., 1993), HOM (Zhang et al., 1997b), and GigaWalk (Baxter et al., 2002). These algorithms use an object-space hierarchy such as an octree or an axis-aligned bounding box (AABB) tree. Each node in the hierarchy is associated with a bounding box that encloses the geometric primitives within the scene-hierarchy below the node (including the node). At run-time, the scene hierarchy is traversed top-down, and the bounding box of the node is tested against the view frustum. If the bounding box is in the view frustum,

the box is tested for visibility against the image representation (Z pyramid, hierarchical occlusion maps, etc). The use of image-space representations enables these algorithms to perform occluder fusion and obtain a good amount of culling in general environments with high depth complexity.

Region-Based Visibility Algorithms

These algorithms pre-compute visibility for a region of space to reduce the run-time overhead (Durand et al., 2000; Schaufler et al., 2000; Wonka et al., 2000).

Durand et al. (Durand et al., 2000) propose a visibility pre-processing algorithm for computing potential visible geometry for volumetric view cells efficiently. The algorithm uses *extended projection* operators for performing conservative occlusion culling with respect to all view-points in a cell. These projection operators are generalizations of occlusion maps for volumetric cells. The approach is image-based and therefore, has the advantage of obtaining combined occlusion effect of multiple occluders. The approach can compute conservative occludee projections and may result in a large potentially-visible sets for some cells.

The approach presented by Wonka et al. [2001] computes the PVS for a region at run-time in parallel with the main rendering pipeline and works well for urban environments. The PVS computation, however, is performed using the *occluder shrinking* algorithm (Wonka et al., 2000) to compute the region-based visibility, which works well only if the occluders are large and volumetric in nature. The method also makes assumptions about the user’s motion.

Most of these algorithms work well for scenes with large or convex occluders. Nevertheless, a trade-off occurs between the quality of the PVS estimation for a region and the memory overhead.

2.1.4 Parallel Rendering

A number of parallel algorithms have been proposed in the literature to render large data sets on shared-memory systems or clusters of PCs. These algorithms include techniques for assigning different parts of the screen to different PCs (Samanta et al., 2000). Other cluster-based approaches include WireGL, which allows a single serial application to drive a tiled display over a network (Humphreys et al., 2001) as well as parallel rendering with k-way replication (Samanta et al., 2001). The performance of these algorithms varies with different environments as well as with the underlying hardware. Most of these approaches are application independent.

Parallel algorithms have also been proposed for interactive ray tracing of volumetric and geometric models on a shared-memory multi-processor system (Parker et al., 1999). A fast algorithm for distributed ray tracing of highly complex models has been described in (Wald et al., 2001).

2.2 Shadow Generation

Shadows refer to the problem of computing visibility of polygons from the light source. In this section, we give a brief overview of previous work on shadow generation. Woo et al. (Woo et al., 1990) give a survey of some of the basic shadowing techniques. We limit ourselves to algorithms that compute hard-edged umbral shadows. In general, shadowing algorithms can be classified as either image-precision or object-precision. A few hybrid combinations have also been proposed.

2.2.1 Image-Precision Methods

These techniques perform visibility computations using discrete image representations to check whether a point is in shadow or not. Popular image-precision techniques for generating shadows are shadow maps (Williams, 1978) and ray tracing.

Shadow maps were introduced by Williams (Williams, 1978) as an image-precision solution for generating shadows. A shadow map is simply a depth map generated from the light's view. In order to determine whether a point lies in shadow, its light-space depth is compared to the depth value stored in the shadow map. Shadow maps can be implemented with standard hardware (Segal et al., 1992; Heidrich and Seidel, 1999), and recent graphics cards have improved support for handling shadow mapping efficiently. By using parabolic projections, shadow maps can be used for hemispherical and omnidirectional light sources (Brabec et al., 2002).

One of the main drawbacks of shadow maps is aliasing. Aliasing can occur when a shadow map pixel projected on the scene subtends more than one pixel in the eye view. There are two main types of aliasing: perspective aliasing and projective aliasing (Stamminger and Drettakis, 2002). Perspective aliasing occurs when a point is much closer to the eye than to the light source. Projective aliasing occurs when the angle formed with a surface normal is greater for the light direction than the view direction. These situations arise often in walkthroughs of complex models with curved objects and wide depth range.

Many techniques have been proposed for handling aliasing of shadow edges. Reeves et al. (Reeves et al., 1987) introduced percentage-closer filtering which improves the appearance of aliased edges by blurring them. In some situations the blurring may be excessive or even undesirable. Brabec et al. (Brabec et al., 2001) applied this filtering for hardware-based shadow map rendering. Fernando et al. (Fernando et al., 2001) presented adaptive shadow maps which are used to increase the effective shadow map resolution in areas where edge aliasing occurs. Unfortunately, adaptive shadow maps require software rendering, which is too slow for interactive rendering of large models. Adaptive shadow maps also use progressive refinement, which may not work well for scenes with a moving light source. Another approach similar to adaptive shadow maps uses multiple shadow maps of varying resolution (Tadamura et al., 2001). Perspective

shadow maps (Stamminger and Drettakis, 2002) ameliorate aliasing by warping the depth buffer in order to allocate more samples near the viewer. Though perspective shadow maps can often reduce perspective aliasing, their performance is highly view dependent and they do not reduce projection aliasing. Silhouette shadow maps (Sen et al., 2003) improve the visual quality by augmenting a shadow map with the location of points on the geometric silhouette. The approach requires the use of fragment programs on GPUs and may be much slower in performance as compared to traditional shadow maps or perspective shadow maps. In addition, the quality of shadows is view dependent and may still involve aliasing artifacts.

Other image-precision methods for shadow generation are based on ray tracing. Many algorithms for fast ray tracing have been proposed on shared-memory multi-processor systems (Parker et al., 1999) as well as on a cluster of PCs (Wald et al., 2001).

2.2.2 Object-Precision Methods

Object-precision approaches avoid the edge-aliasing problem by computing exact shadow boundaries. These approaches include projection techniques that calculate shadow boundaries on the scene polygons. Atherton et al. (Atherton et al., 1978) clip the scene polygons against each other from the light view. The resulting clipped polygons, representing the lit surfaces, are attached to the original polygons as surface details. Blinn (Blinn, 1988) rendered shadows by projecting the vertices of an occluder object onto the plane of a receiver polygon and used the resulting polygons to modulate the surface color. In practice, however, these techniques do not scale well to large models.

One of the most popular object-precision techniques is the shadow volume algorithm introduced by Crow (Crow, 1977). A shadow volume is the set of points that lie in shadow behind a shadow-caster. For a polygonal shadow-caster, the shadow volume is

a semi-infinite frustum extending away from the edges of the polygon to infinity. The shadow-volume algorithm checks whether a particular point is in shadow by counting the crossings with shadow-frusta polygons on any ray extending away from the point. Bergeron (Bergeron, 1985) generalized shadow volumes for non-manifold objects and non-planar polygons. BSP trees have been used to represent shadow volumes (Chin and Feiner, 1989; Chrysanthou and Slater, 1995). These techniques do not work well with dynamic lights because the entire tree has to be rebuilt when the light source moves. Heidmann (Heidmann, 1991) showed that shadow volumes can be implemented in hardware by using the stencil buffer to count crossings. Recently, techniques have been proposed to ensure that hardware shadow volumes are not clipped “open” by the near and far clipping planes (Everitt and Kilgard, 2002). Brabec and Seidel (Brabec and Seidel, 2003) described an algorithm for fast shadow-volume computation using the graphics hardware for silhouette-edge computation. The enhanced robustness of the algorithm has led to shadow volumes’ becoming increasingly more popular in games (e.g., Doom-3).

Shadow volumes, however, may not scale well for complex models. The number of shadow polygons can be extremely large. A common configuration in walkthroughs is a light source overhead with geometry such as beams or trusses above the viewer. The shadow-frustum polygons created by the overhead geometry may fill the whole screen, yet the shadows they define may be very small. In the presence of many large shadow volumes, the application will quickly become fill-bound.

In order to reduce the fill generated by shadow volumes, Lengyel (Lengyel, 2002) proposed the use of a scissor test for restricting shadow volumes to within the light bounds. McGuire et al. (McGuire et al., 2003) improved upon Lengyel’s algorithm by adding culling and using the depth-bounds test (D03 depthbounds, 2003) to restrict shadow-volume rendering further. Chan and Durand (Chan and Durand, 2004) propose a hybrid algorithm that combines shadow maps and shadow volumes to reduce fill. They

render a shadow map to identify shadow boundaries and render shadow volumes only in these areas. Aila and Möller (Aila and Akenine-Möller, 2004) perform shadow-volume calculations on coarse tiles in screen space to determine which tiles contain shadow boundaries and then render shadow volumes only in these tiles. These approaches rely on existing or proposed culling hardware to avoid unnecessary rendering. Lloyd et al. (Lloyd et al., 2004) proposed an algorithm that reduces the size and complexity of the rendered shadow volume.

2.2.3 Hybrid Approaches

Some combinations of object-space and image-space techniques have been proposed for shadow generation and related computations. Brotman and Badler (Brotman and Badler, 1984) combined shadow volumes with a software-based, depth-buffered, tiled renderer to generate soft shadows. McCool (McCool, 2000) extracts edges from a shadow map to create shadow volumes. While the technique replaces aliased edges with sharp edges, it does not replace the details lost due to the limited resolution of the shadow map. Udeshi and Hansen (Udeshi and Hansen, 1999) presented an improved shadow-volume algorithm using multiple CPUs and graphics processors on a shared-memory architecture, but they only rendered relatively small indoor scenes.

2.3 Interference Detection

The problem of interference detection is related to the computation of portions of object boundaries that overlap with other objects in the scene. As it involves overlap detection, and visibility computations can be used to compute overlapping regions, many researchers have proposed visibility-based algorithms for interference detection (Baciu and Wong, 2002; Hoff et al., 2001; Heidelberger et al., 2003; Govindaraju et al., 2003b). In this section, we briefly survey interference-detection algorithms and primarily focus

on visibility-based interference-detection algorithms.

Interference-detection algorithms can be classified broadly into three categories: object-space algorithms, image-space algorithms and hybrid algorithms. Most of these algorithms operate in two phases: the “broad phase,” in which collision culling is performed to reduce the number of pairwise tests, and the “narrow phase,” in which the pairs of objects in proximity are checked for collision (Cohen et al., 1995; Hubbard, 1993).

2.3.1 Object-Space Algorithms

Object-space algorithms use spatial data structures to accelerate interference computations. These spatial data structures include spatial-partitioning structures and bounding-volume hierarchies. Some of the commonly used bounding-volume hierarchies include sphere trees (Hubbard, 1993; Quinlan, 1994), AABB trees (Beckmann et al., 1990; Ponamgi et al., 1997), OBB trees (Gottschalk et al., 1996a; Barequet et al., 1996), k-DOP trees (Held et al., 1996; Klosowski et al., 1998), etc. At run-time, the hierarchy is traversed, and bounding volume representations of nodes of the hierarchy are used to cull away portions of objects that are not in close proximity. Typically, these representations are built in a pre-processing stage to accelerate run-time queries. In practice, they work well for rigid objects.

Efficient algorithms for handling large environments consisting of multiple moving objects have also been designed. These techniques reduce the number of pairwise collision checks by using spatial subdivision algorithms or checking whether the bounding boxes of the objects overlap (Cohen et al., 1995; Baciú et al., 1998). (Cohen et al., 1995) propose a sweep-and-prune algorithm for pruning objects that are not in close proximity. For each object, its AABB is projected along the three world-space axes and is tested for overlap with the bounding-box projections of remaining objects. If the AABB is not overlapping with any object along any of the three axes, then it does not collide with

other objects in the environment. The overlap operation can be performed efficiently on all the objects by insert sorting the intervals of each object's AABB along the axes. Using temporal coherence, the expected running time of insertion sort is reduced to $O(n)$, where n is the number of objects in the scene. In practice, the algorithm works well for large environments composed of *rigid* objects.

For deformable models, the overhead of recomputing the hierarchy on the fly can be quite significant (Baciu and Wong, 2002; Hoff et al., 2001). More recently, (Larsson and Akenine-Moller, 2001) proposed different bounding-volume trees suitable for fast update in deforming objects. For fast updates, they maintain AABBs in bounding-volume trees and use them for collision culling. However, in complex simulations with many objects in close proximity, AABBs may not provide effective culling as they are conservative, and thereby affect the overall performance.

2.3.2 Image-Space Algorithms

Many image-space algorithms have been designed on graphics processors for interference and collision computations (Baciu et al., 1998; Baciu and Wong, 2002; Heidelberger et al., 2003; Hoff et al., 2001; Knott and Pai, 2003; Myszkowski et al., 1995; Rossignac et al., 1992; Shinya and Forgue, 1991; Vassilev et al., 2001). These algorithms require no pre-processing and therefore are well suited for handling deformable models. Most of these algorithms use visibility computations for detecting overlapping regions between objects. Objects are rendered from a view-point, and 2-D or 2.5-D overlaps are detected in image-space.

(Lombardo et al., 1999) use view-frustum culling to prune interactions of a surgical tool with the organs in virtual surgery. Many image-space algorithms detect 2-D overlaps using color buffer (Baciu et al., 1998; Baciu and Wong, 2002; Vassilev et al., 2001; Rossignac et al., 1992). These algorithms use color-blending operations to encode the overlapping polygons in color buffer. Few algorithms use the stencil buffer for detecting

overlaps (Hoff et al., 2001; Knott and Pai, 2003). Knott and Pai (Knott and Pai, 2003) use original objects as shadow volumes, setting stencil in portions of the screen that are overlapping with other objects. The algorithm is approximate and works only on closed objects. Algorithms using depth information in addition to 2-D overlap information have also been proposed (Knott and Pai, 2003; Heidelberger et al., 2003; Govindaraju et al., 2003b). Heidelberger et al. (Heidelberger et al., 2003) compute layer depth images (LDIs) on the GPU, use the LDIs for explicit computation of the intersection volumes between two closed objects, and perform vertex-in-volume tests.

Many of these algorithms are limited to closed objects or involve frame-buffer readbacks or both. Frame-buffer readbacks can be slow on current graphics systems, as they involve graphics pipeline stalls, and are limited by the bandwidth from GPU to CPU (Knott and Pai, 2003; Govindaraju et al., 2003b). Graphics-pipeline stalls affect the rendering throughput and thus reduce the performance of the underlying algorithms. A major drawback of current hardware-accelerated image-space algorithms is the inaccuracy in collision computations. These inaccuracies occur due to the limited viewport resolution and frame-buffer precision on GPUs, resulting in some missed collision events. Such errors can often lead to simulations that are not physically convincing.

There exist software implementations for reliable interference detection using fat edges¹ and read back multiple depth layers (Rossignac et al., 1992), but they work well only on pairs of objects involving few contacts. Also, Rossignac et al. (Rossignac et al., 1992) does not address the issue of aliasing in the depth buffer. This limitation is addressed in (Raskar and Cohen, 1999) and is used for rendering image-precision silhouette edges. Raskar and Cohen (Raskar and Cohen, 1999) fatten the back-facing polygons for rendering silhouette edges. The front-facing polygons are not fattened, as the technique only *renders* silhouette edges. As some polygons are not fattened, the technique described in (Raskar and Cohen, 1999) may miss interferences due to limited

¹Fat edges are edges with linewidth of more than one pixel.

image precision.

2.3.3 Hybrid Algorithms

Hybrid algorithms combine some of the benefits of the object-space and image-space approaches. (Hoff et al., 2001) perform coarse geometric localization by computing rectangular regions of space that contain potential intersections and then perform image-based interference computations within these rectangular regions. Kim et al. (Kim et al., 2002b) compute the closest distance from a point to the union of convex polytopes using the GPU, refining the answer on the CPU. The accuracy of these algorithms is limited by the viewport resolution.

Chapter 3

Visibility Computations: Interactive Walkthroughs

3.1 Introduction

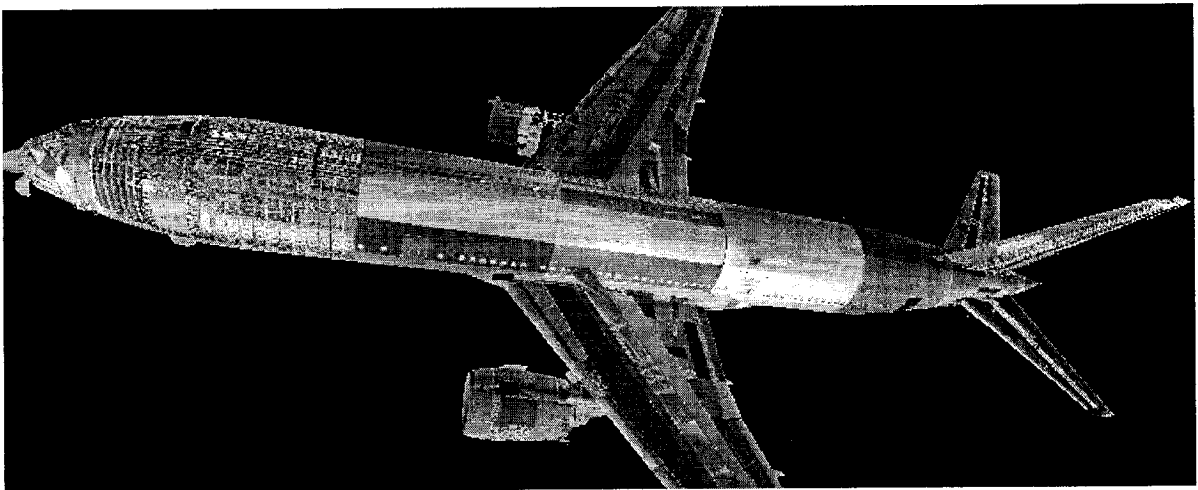


Figure 3.1: *This figure shows a view of the Boeing 777 model. The model is designed using more than 470 million polygons.*

Many CAD and virtual-reality applications often involve large geometric models composed of millions of primitives. For example, a Boeing 777 airplane model is designed using 470 million polygons (shown in Fig. 3.1). These models are composed of several complex piping structures and exhibit high visual complexity. In this chapter,

we describe a new *parallel* occlusion culling algorithm for rendering large geometric environments at interactive frame rates. Our visibility culling algorithm “*occlusion switch*” uses two GPUs. During each frame, one GPU renders the occluders and computes an occlusion representation, while the second GPU performs culling in parallel using image-space occlusion queries. In order to avoid any depth-buffer readbacks and to perform significant occlusion culling, the two GPUs switch their roles between successive frames. The visible primitives computed by the occlusion switch are rendered in parallel on a third GPU. The algorithm utilizes frame-to-frame coherence to compute occluders for each frame as well as to lower the bandwidth or communication overhead between different GPUs.

We have combined the occlusion culling algorithm with static levels-of-detail (LODs) and used it for interactive walkthrough of complex environments. The static LODs are computed as a pre-process. We present a novel clustering algorithm for generating a unified scene-graph hierarchy, which is used for generating LODs and performing occlusion culling.

The rest of the chapter is organized in the following manner. In Section 3.2, we present our occlusion culling algorithm “*occlusion switch*”, and analyze its bandwidth requirements. In Section 3.3, we describe the scene representation and preprocessing steps including hierarchy computation. In Section 3.4, we combine our occlusion culling algorithm with pre-computed levels-of-detail, and use it to render large environments. We describe its implementation and highlight its performance on three complex environments in Section 3.5. Finally, we provide an analysis of our algorithm and describe its limitations.

The clustering algorithm described in this chapter is presented in (Baxter et al., 2002). Details of the occlusion culling algorithm are presented in¹ (Govindaraju et al., 2002; Govindaraju et al., 2003c).

¹Joint work with William Baxter, Avneesh Sud and Sung-Eui Yoon

3.2 Interactive Occlusion Culling

In this section, we present occlusion switches and use them for visibility culling. The resulting algorithm uses multiple graphics processing units with image-space occlusion queries.

3.2.1 Occlusion Representation and Culling

An occlusion culling algorithm has three main components. These include

1. computing a set of occluders that correspond to an approximation of the visible geometry,
2. computing an occlusion representation, and
3. using the occlusion representation to cull primitives that are not visible.

Different culling algorithms perform these steps either explicitly or implicitly. We use an image-based occlusion representation because it is able to perform “occluder fusion” on possibly disjoint occluders (Zhang et al., 1997a). Some of the well-known image-based hierarchical representations include HZB (Greene et al., 1993) and HOM (Zhang et al., 1997a). The current GPUs, however, do not support these hierarchies entirely in the hardware. Many two-pass occlusion culling algorithms rasterize the occluders, read back the depth buffer, and build the hierarchies in software (Baxter et al., 2002; Greene et al., 1993; Zhang et al., 1997a).

However, reading back a high resolution depth buffer can be slow on current PC architectures, as described in Section 1.5. Moreover, constructing the hierarchy in software incurs additional overhead.

We utilize the hardware-based occlusion queries that are becoming common on current GPUs. These queries scan-convert the specified primitives (e.g., bounding boxes) to check whether the depth of any pixel changes. Different queries vary in

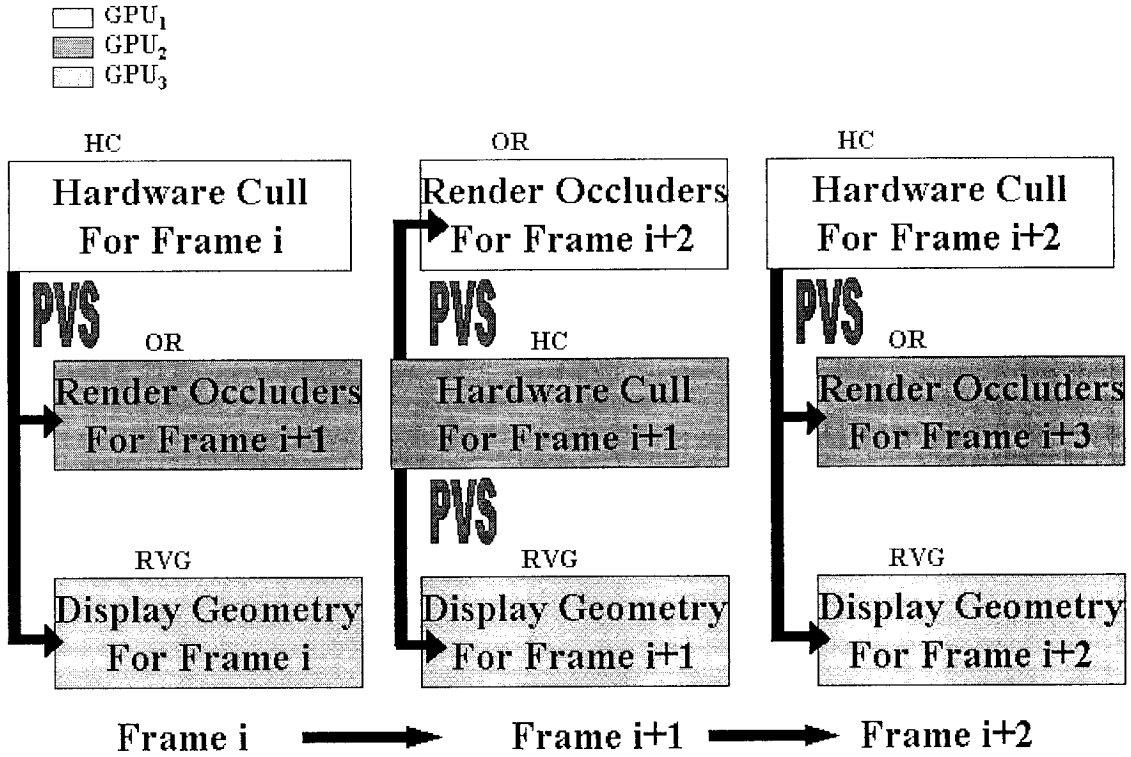


Figure 3.2: *System Architecture: Each color represents a separate GPU. Note that GPU₁ and GPU₂ switch their roles each frame with one performing hardware culling and other rendering occluders. GPU₃ is used as a display client.*

their functionality. Some of the well-known occlusion queries based on the OpenGL culling extension include the HP_Occlusion_Query (http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt) and the NVIDIA OpenGL extension GL_NV_occlusion_query (http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt). These queries can sometimes stall the pipelines while waiting for the results. As a result, we use a specific GPU during each frame to perform only these queries.

Our algorithm uses the visible geometry from frame i as an approximation to the occluders for frame $i + 1$. The occlusion representation implicitly corresponds to the depth buffer after rasterizing all these occluders. The occlusion tests are performed using hardware-based occlusion queries. The *occlusion switches* are used to compute

the occlusion representation and perform these queries.

3.2.2 Occlusion Switch

An occlusion switch takes the view parameters for frame $i + 1$ as input and transmits the potential visible set and view parameters for frame i as the output to the renderer. An occlusion switch is composed of two GPUs, which perform the following functions, with each function running on a separate GPU in parallel:

- **Computing Occlusion Representation (OR):** Rendering the occluders in order to compute the occlusion representation. The occluders for frame $i + 1$ correspond to the visible primitives from frame i .
- **Hardware Culling (HC):** Enabling the occlusion-query state on the GPU and rendering the bounding boxes corresponding to the scene geometry. Using the image-space occlusion query to determine the visibility of each bounding box and compute the PVS. We disable modifications to the depth buffer while performing these queries.

During a frame, each GPU in the occlusion switch performs either OR or HC, and at the end of the frame the two GPUs interchange their functions. The depth buffer computed by OR during the previous frame is used by HC to perform the occlusion queries during the current frame. The visible nodes computed by HC correspond to the PVS. The PVS is rendered in parallel on a third GPU and is used by the OR for the next frame to compute the occlusion representation. The architecture of the overall system is shown in Fig. 3.2. The overall occlusion algorithm involves no depth buffer readbacks from the GPUs.

3.2.3 Culling Algorithm

The occlusion culling algorithm uses an occlusion switch for computing the PVS and renders them in parallel on a separate GPU. GPU_1 and GPU_2 constitute the occlusion switch, and GPU_3 is used to render the visible primitives (RVG). In an occlusion switch, the GPU performing HC requires OR for occlusion tests. We circumvent the problem of transmitting occlusion representation from the GPU generating OR to the GPU performing hardware cull tests by “switching” their roles between successive frames, as shown in Fig. 3.2. For example, GPU_1 is performing HC for frame i and sending visible nodes to GPU_2 (to be used to compute OR for frame $i + 1$) and GPU_3 (to render visible geometry for frame i). For frame $i + 1$, GPU_2 has previously computed OR for frame $i + 1$. As a result, GPU_2 performs HC, GPU_1 generates the OR for frame $i + 2$ and GPU_3 displays the visible primitives.

3.2.4 Incremental Transmission

The HC process in the occlusion culling algorithm computes the PVS for each frame and sends it to the OR and the RVG. In order to minimize the communication overhead, we exploit frame-to-frame coherence in the list of visible primitives. Each GPU keeps track of the visible nodes in the previous frame. The GPU performing HC uses this list and only transmits the changes to the other two GPUs. The GPU performing HC sends the visible nodes to OR and RVG, and therefore it has information related to the visible set on HC. Moreover, the other two processes, OR and RVG, maintain the visible set as they receive visible nodes from HC. In order to reduce the communication bandwidth, we transmit only the difference in the visible sets for the current and previous frames. Let V_i represent the potential visible set for frame i and $\delta_{j,k} = V_j - V_k$ be the difference between the two sets. During frame i , HC transmits $\delta_{i,i-1}$ and $\delta_{i-1,i}$ to OR and RVG,

respectively. We reconstruct V_i at OR and RVG based on the following formulation:

$$V_i = (V_{i-1} - \delta_{i-1,i}) \cup \delta_{i,i-1}.$$

We expect that the size, in most interactive applications, of the set $\delta_{i-1,i} \cup \delta_{i,i-1}$ will be much smaller than that of V_i due to frame-to-frame coherence.

3.2.5 Bandwidth Requirements

We now discuss the bandwidth requirements of our algorithm for a distributed implementation on three different graphics systems (PCs). Each graphics system consists of a single GPU, and the three PCs are connected using a network. In particular, we map each node of the scene by the same node identifier across the three different graphics systems. We transmit these integer node identifiers across the network from the GPU performing HC to each of the GPUs performing OR and RVG. This procedure is more efficient than sending all the triangles that correspond to the node, as it requires a smaller bandwidth per visible node (i.e. 4 bytes per node). So, if the number of visible nodes is n , then the GPU performing HC must send $4n$ bytes per frame to each OR and RVG client. Here n refers to the number of visible objects and not the visible polygons. We can reduce the header overhead by sending multiple integers in a packet. This process, however, can introduce some extra latency in the pipeline due to buffering. The size of view parameters is 72 bytes; consequently, the bandwidth requirement per frame is $8n + nh/b + 3(72 + h)$ bytes, where h is the size of the header in bytes and buffer size b is the number of node identifiers in a packet. If the frame rate is f frames per second, the total bandwidth required is $8nf + nhf/b + 216f + 3hf$. If we send the visible nodes by incremental transmission, then n is equal to the size of $\delta_{i,i-1} \cup \delta_{i-1,i}$.

3.3 Scene Representation

In this section, we present an object-clustering based algorithm for computing a scene-graph representation of the geometric data set automatically.

CAD data sets often consist of a large number of objects that are organized according to a functional, rather than a spatial, hierarchy. By “object” we mean simply the lowest level of organization in a model or model data structure above the primitive level. The size of objects can vary dramatically in CAD data sets. For example, in the Power Plant model a large pipe structure, which spans the entire model and consists of more than 6 million polygons, is one object; a relatively small bolt with 20 polygons is another. Our rendering algorithm computes LODs, selects them, and performs occlusion culling at the object level; therefore, the criteria used for organizing primitives into objects has a serious impact on the performance of the system. Our first step, then, is to redefine objects in a data set based on criteria that will improve performance.

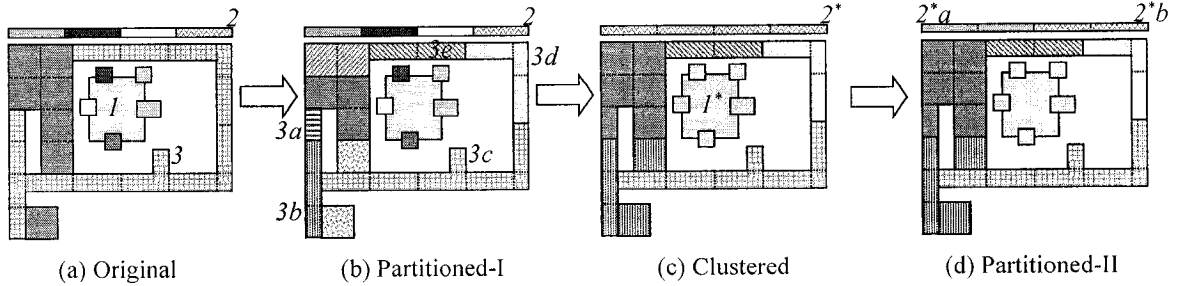


Figure 3.3: *Our clustering and partitioning process applied to a 2-D example. Each different color represents a different object at the end of a stage. (a) The model’s original objects. This object distribution captures a number of features common in CAD models in which objects are defined by function rather than by location. (b) The initial partitioning stage splits objects with large bounding boxes. This prevents objects like 3, whose initial bounding box intersects most of the others, from causing clustering to generate just one large cluster. (c) After clustering, the group of the small objects around 1 have all been merged to form 1^* . The row of objects, 2, has been merged into one cluster, 2^* , as well, but one which has a poor aspect ratio. (d) The final partition splits 2^* into two separate objects.*

3.3.1 Unified Scene Hierarchy

Our rendering algorithm performs occlusion culling using two rendering processes: the OR process renders occluders in order to create an occlusion representation for culling, and the RVG process renders the objects that are deemed visible by the HC process. Given this approach, we could consider using a separate representation for occluders in OR than for displayed objects in RVG (Hudson et al., 1997a; Zhang et al., 1997a). Using different representations would have the advantage of allowing different criteria for partitioning and clustering for each hierarchy. Moreover, it would give us the flexibility of using a different error metric for creating simplified occluders, one optimized to preserve occlusion rather than visual fidelity.

Despite these potential advantages, we used a single unified hierarchy for occlusion culling and levels-of-detail-based rendering. A single hierarchy offers the following benefits:

- **Simplicity:** A single representation leads to a simpler algorithm.
- **Memory And Preprocessing Overhead:** A separate occluder representation would increase the storage overhead and increase the overall preprocessing cost. This increase can be significant for gigabyte data sets, depending upon the type of occluder representation used.
- **Conservative Occlusion Culling:** Our rendering algorithm treats the visible geometry from the previous frame as the occluder set for the current frame. In order to guarantee conservative occlusion culling, it is sufficient to ensure that exactly the same set of nodes and LODs in the scene graph is used by each process. Ensuring conservative occlusion culling when different representations are used is more difficult.

Criteria for Hierarchy

A good hierarchical representation of the scene graph is crucial for the performance of occlusion culling and the overall rendering algorithm. We use the same hierarchy for

view-frustum culling, occluder selection, occlusion tests on potential occludees, hierarchical simplification, and LOD selection. Though there has been considerable work on spatial-partitioning and bounding-volume hierarchies, including top-down and bottom-up strategies and spatial clustering, none of them seem to have addressed all the characteristics desired by our occlusion culling algorithm. These include good spatial localization, object size, balanced hierarchy, and minimal overlap between the bounding boxes of sibling nodes in the tree.

Bottom-up hierarchies lead to better localization and higher fidelity LODs. However, it is harder to use bottom-up techniques to compute hierarchies that not only are balanced but also have minimal spatial overlap between the nodes. On the other hand, top-down schemes are better at ensuring balanced hierarchies and bounding boxes with little or no overlap between sibling nodes. Given their respective benefits, we use a hybrid approach that combines both top-down partitioning and hierarchy construction with bottom-up clustering.

3.3.2 Hierarchy Generation

In order to generate uniformly-sized objects, our pre-processing algorithm first redefines the objects using a combination of partitioning (Private Communication with Sud, 2003) and clustering algorithms (see Fig. 3.3). The partitioning algorithm splits large objects into multiple objects. The clustering step groups objects with low polygon counts based on their spatial proximity. The combination of these steps results in a redistribution of geometry with good localization and emulates some of the benefits of pure bottom-up hierarchy generation. The overall algorithm proceeds as follows:

1. Partition large objects into subobjects in the initial database (top-down)
2. Organize disjoint objects and subobjects into clusters (bottom-up)
3. Partition again to eliminate any uneven spatial clusters (top-down)

4. Compute an AABB bounding-volume hierarchy on the final redefined set of objects (top-down).

Next, we present the algorithms for clustering and partitioning in detail.

Partitioning Objects

We subdivide large objects—based on their sizes, aspect ratios, and polygon counts—into multiple subobjects, since long, thin objects with large bounding boxes are less likely to be occluded. The algorithm proceeds as follows:

1. Check whether an object meets the splitting criteria:
 - the number of triangles is above a threshold t_1 , and
 - the size (bounding-box diagonal) is greater than a threshold s_1 , or
 - the ratio of largest dimension of bounding box to smallest dimension is above a threshold r_1 .
2. Partition the object along the longest axis of the bounding box. If that results in an unbalanced partition, choose the next longest axis.
3. Split the child objects recursively in the same manner.

See Section 3.5.1 for the parameter values used for this algorithm and the subsequent stages of our preprocess.

Clustering

Many approaches for clustering disjoint objects are based on spatial partitioning by such means as adaptive grids or octrees. These partitioning approaches, however, may not work well for complex, irregular environments composed of a number of small and large objects. Rather, we present an object-space clustering algorithm by extending

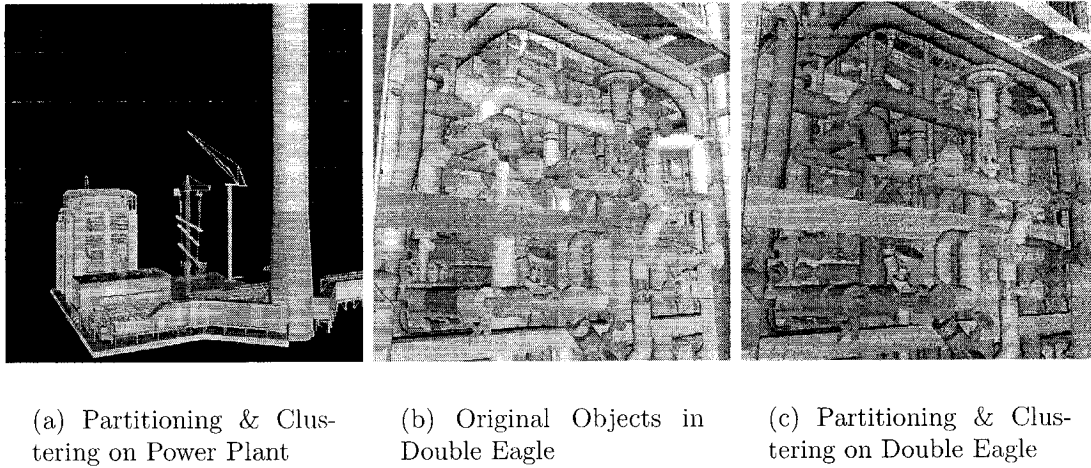


Figure 3.4: *The image on the left shows the application of the partitioning and clustering algorithm to the Power Plant model. The middle image shows the original objects in the Double Eagle tanker model with different colors. The right image shows the application of the clustering algorithm to the same model. Each cluster is shown with a different color.*

a computer-vision algorithm for image segmentation (Felzenszwalb and Huttenlocher, 1998). The algorithm uses minimum spanning trees (MST) for representing the clusters. It uses local spatial properties of the environment to incrementally generate clusters that represent global properties of the underlying geometry. The resulting clusters are neither too coarse nor too fine. The algorithm imposes this criterion by ensuring that it combines two clusters only if the internal variation in each cluster is greater than its external variation (by using the Hausdorff metric). In each cluster, the maximum-weight edge (which denotes the internal variation) of the MST representing the cluster denotes the maximum separation between any two “connected” objects in the cluster. A minimum-weight edge connecting two different clusters (which denotes the external variation between the two clusters) estimates the separation between objects of one cluster with those of the other. In other words, it denotes the minimum radius of dilation necessary to connect at least one point of one cluster to one point in another. The algorithm is similar to *Kruskal’s* algorithm (Kruskal, 1956) for generating a forest of minimum spanning trees. The overall algorithm proceeds as follows:

1. Construct a graph $G(V, E)$ of the environment with each object represented by

a vertex in the graph. Construct an edge between two vertices if the distance between the two vertices (objects) is less than a threshold D . The distance function is defined as the shortest distance between the bounding boxes of the two objects. If the two boxes overlap, then the distance is zero.

2. Sort E into $\pi = (o_1, o_2, \dots, o_k)$, $o_i \in E$, $i = 1, \dots, k$ in a non-decreasing order based on edge weights, $w(o_i)$. Start with a forest F^0 where each vertex v_i represents a cluster.
3. Repeat Step 4 for the set of edges $o_q = o_1, \dots, o_k, k = \|E\|$.
4. Construct forest F^q from F^{q-1} as follows. Let $o_q = (v_i, v_j)$, (i.e., edge o_q connects vertices v_i and v_j). If there is no path from v_i to v_j in F^{q-1} and $w(o_q)$ is small compared to the internal variation of components containing v_i and v_j , and if the bounding volume of the resultant component is less than a maximum volume threshold, then add o_q to F^{q-1} in order to obtain F^q , otherwise add nothing. Mathematically, if $C_i^{q-1} \neq C_j^{q-1}$ and $w(o_q) \leq \mathcal{MI}(C_i^{q-1}, C_j^{q-1})$, then $F^q = F^{q-1} \cup o_q$ where C_i^{q-1} denotes the cluster containing vertex v_i in F^{q-1} and C_j^{q-1} is the cluster containing vertex v_j ; else $F^q = F^{q-1}$, where

$$\mathcal{MI}(C_1, C_2) = \min(\mathcal{I}(C_1) + K/\|C_1\|, \mathcal{I}(C_2) + K/\|C_2\|)$$

and

$$\mathcal{I}(C) = \max_{e \in MST(C, E)} w(e) \quad (3.1)$$

The $K/\|C_i\|$ terms bias the results toward clusters of cardinality bounded by $O(K)$, where K is user specified. We set a maximum volume threshold V in order to ensure final clusters are not too large in size. The algorithm is reasonably fast in practice and generates good spatial clusters. Fig. 3.4 shows its application to the Power Plant and

Double Eagle models.

Overall, the clustering algorithm successfully merges small objects into larger groups with good localization. It improves the performance of the culling algorithm as well as the final-image fidelity.

Partitioning Clusters

Finally, we repartition clusters with unevenly distributed geometries, splitting objects about their centers of mass. This is similar to the partitioning algorithm presented in Section 3.3.2; however, we use higher thresholds for the size and triangle count in order to avoid splitting clusters back into small objects, but tighter bounds for the aspect ratio in order to force splitting of uneven clusters.

Hierarchy Generation

We compute a standard AABB bounding-volume hierarchy in a top-down manner on the set of redefined objects generated after clustering and partitioning. The bounding boxes are assigned to left and right nodes of the hierarchy using their geometric centers in order to avoid overlap between the nodes. The redefined objects become the leaf nodes in the AABB hierarchy.

3.3.3 HLOD Generation

Given the above scene graph, the algorithm computes a series of LODs for each node. The HLODs are computed in a bottom-up manner. The HLODs of the leaf nodes are standard static LODs, while the HLODs of intermediate nodes are computed by combining the LODs of the nodes with the HLODs of node’s children. We use a topological-simplification algorithm for merging disjoint objects (Erikson and Manocha, 1999).

The majority of the pre-computation time is spent in LOD and HLOD generation.

The HLODs of an internal node depend only on the LODs of the children, so by keeping only the LODs of the current node and its children in main memory, HLOD generation is accomplished within a small memory footprint. Specifically, the memory usage is given by

$$\begin{aligned} \text{main_memory_footprint} \leq & \text{sizeof}(AABBHierarchy) \\ & + \max_{N_i \in SG} (\text{sizeof}(N_i) + \\ & \sum_{C_j \in Child(N_i)} \text{sizeof}(C_j)) \end{aligned}$$

where SG denotes the scene graph and N_i is a node in the scene graph.

3.3.4 HLODs as Hierarchical Occluders

Our occlusion culling algorithm uses LODs and HLODs of nodes as occluders to compute the occlusion representation. They are selected based on the maximum screen-space pixel-deviation error on object silhouettes.

The HLODs our rendering algorithm uses for occluders can be thought of as “hierarchical occluders.” A hierarchical occluder associated with a node i is an approximation of a group of occluders contained in the subtree rooted at i . The approximation provides a lower-polygon-count representation of a collection of object-space occluders. It can also be regarded as object-space occluder fusion.

3.4 Interactive Display

In this section, we present our overall rendering algorithm for interactive display of large environments. We use the occlusion culling algorithm described above and combine it with pre-computed static levels-of-detail (LODs) in order to render large

environments. We represent our environment using a scene graph, as described in Section 3.3.3. We describe our occlusion culling algorithm and highlight many optimizations used for improving the overall performance.

```

HardwareCull(Camera *cam)
1  queue = root of scene graph
2  disable color mask and depth mask
3  while( queue is not empty)
4  do
5      node = pop(queue)
6      visible= OcclusionTest(node)
7      if(visible)
8          if(error(node) < pixels of error)
9              Send node to OR and RVG
10         else
11             push children of node to end of queue
12         endif
13     end if
14 end do

```

ALGORITHM 3.4.1: *Pseudocode for Hardware cull (HC). OcclusionTest renders the bounding box and returns either the number of visible pixels or a boolean, depending upon the implementation of the query. The function error(node) returns the screen-space projection error of the node. Note that if the occlusion test returns the number of visible pixels, we could use it for computing the level at which it must be rendered.*

3.4.1 Culling Algorithm

At run-time, we traverse the scene graph and cull away portions of the geometry that are not visible. The visibility of a node is computed by rendering its bounding box, comparing the rasterized depth values against the occlusion representation and querying whether the bounding box is visible or not. Testing the visibility of a bounding box is a fast and conservative way to reject portions of the scene that are not visible. If the bounding box of the node is visible, we test whether any of the LODs or HLODs associated with that node meet the pixel-deviation error bound. If one of the LODs or HLODs is selected, we include that node in the PVS and send it to the GPU performing OR for the next frame as well as to the GPU performing RVG for the current frame. If

the node is visible and none of the HLODs associated with it satisfy the simplification error bound, we traverse down the scene graph and apply the procedure recursively on each node. On the other hand, if the bounding box of the node is not visible, we do not render that node or any node in the subtree rooted at the current node.

The pseudocode for the algorithm is described in Algorithm 3.4.1. The image-space occlusion query is used to perform view-frustum culling as well as occlusion culling on the bounding volume.

3.4.2 Occluder-Representation Generation

At run-time, if we are generating OR for frame $i + 1$, we receive camera $i + 1$ from RVG and set the camera parameters. We also clear the depth buffer of OR. While OR receives nodes from the GPU performing HC, we render them at the appropriate level-of-detail. An end-of-frame identifier is sent from HC to notify the other GPUs that no more nodes need to be rendered for this frame.

3.4.3 Occlusion-Switch Algorithm

We now describe the algorithm for the “switching” mechanism described in Section 3.2. The two GPUs involved in the occlusion switch interchange their roles of performing HC and generating OR. We use the algorithms described in Sections 3.4.1 and 3.4.2 to perform HC and OR, respectively. The pseudocode for the resulting algorithm is shown in Algorithm 3.4.2.

3.4.4 Render Visible Geometry

The display client, RVG, receives the camera for the current frame from HC. In addition, it receives the visible nodes in the scene graph and renders them at the appropriate level-of-detail. Moreover, the display client transmits the camera information


```

1  if GPU is generating OR
2      camera=grabLatestCam()
3  end if
4  Initialize the colormask and depth mask to true.
5  if GPU is performing HC
6      Send Camera to RVG
7  else /*GPU needs to render occluders */
8      Clear depth buffer
9  end if
10 Set the camera parameters
11 if GPU is performing HC
12     HardwareCull(camera)
13     Send end of frame to OR and RVG
14 else /* Render occluders */
15     int id= end of frame +1 ;
16     while(id!=end of frame)
17     do
18         id=receive node from HC
19         render(id, camera);
20     end do
21 end if
22 if GPU is performing HC
23     do OR for next frame
24 else
25     do HC for next frame
26 end if

```

ALGORITHM 3.4.2: *The main algorithm for the implementation of occlusion switch. Note that we send the camera parameters to the RVG client at the beginning of HC (on line 6) in order to reduce latency.*

to the GPUs involved in occlusion switch based on user interaction. The colormask and depthmask are set to *true* during initialization.

3.4.5 Incremental Traversal and Front Tracking

The traversal of the scene graph defines a cut that can be partitioned into a visible front and an occluded front as shown in Fig. 3.6.

- **Visible Front:** composed of all the visible nodes in the cut. In addition, each node belonging to the visible front satisfies the screen-space error metric while its parent does not.

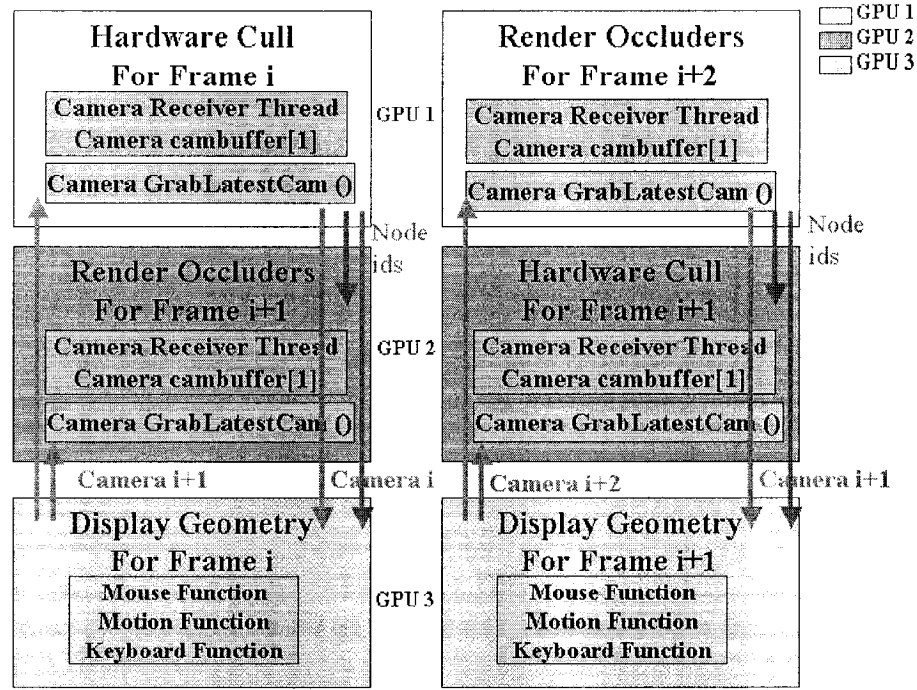


Figure 3.5: *System Overview: Each color represents a separate GPU, with GPU₁ and GPU₂ forming a switch and GPU₃ as the display client. GPU₁ and GPU₂ each have a camera-receiver thread; receive camera parameters when the client transmits them due to user's motion; and store those in a camera buffer of size one. The GPU performing OR takes the latest camera from this thread as the camera position for the next frame. Notice that, in this design, the GPU performing HC exhibits no latency in receiving the camera parameters.*

- **Occluded Front:** composed of all the occluded nodes in the cut. Also, an occluded node may not satisfy the screen-space error metric.

We reduce the communication overhead by keeping track of the visible and occluded fronts from the previous frame at each GPU. Each node in the front is assigned one of the following states:

- **Overrefined:** Both the node and its parent satisfy the silhouette deviation metric in screen space.
- **Refined:** The node satisfies the silhouette deviation metric while the parent does not.
- **Underrefined:** The node does not satisfy the silhouette deviation metric.

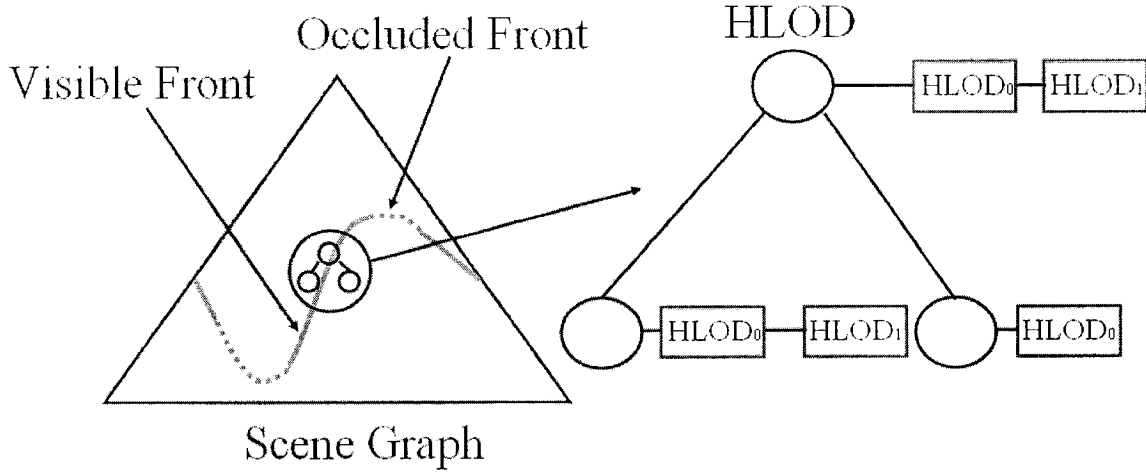


Figure 3.6: This figure shows a cut defined by the traversal of the scene graph using our culling algorithm. The cut is further decomposed into visible and occluded fronts. Each front is represented using the following colors: orange: visible front; and gray: occluded front

Each node in the front is updated depending upon its state. If the node is *Overrefined*, we traverse up the scene graph to reach a parent node which is *Refined*. If the node is *Underrefined*, we traverse down the scene graph generating a set of *Refined* children nodes. At the beginning of each frame, both OR and RVG update the state of each node in the visible front before rendering it.

We also render each node in $\delta_{i,i-1}$ at OR and RVG. At the end of the frame, the visible nodes for the current frame are reconstructed as described in Section 3.2.4. The update of the state of each node is important for maintaining the conservative nature of the algorithm.

At the GPU performing HC, we maintain the occluded front in addition to the visible front of previous frame. Doing so enables us to compute $\delta_{i,i-1}$ efficiently by performing culling on the occluded front before the visible front. A node in the occluded front is refined only if it is in the *Overrefined* state. Each of the occluded fronts and visible fronts is refined before performing the culling algorithm on the refined fronts. Moreover, the nodes in $\delta_{i,i-1}$ are a part of the refined occluded front.

3.4.6 Optimizations

We use a number of optimizations to improve the performance of our algorithms, including

- **Multiple Occlusion Tests:** Our culling algorithm performs multiple occlusion tests using `GL_NV_occlusion_query`; this avoids immediate readback of occlusion identifiers, which can stall the pipeline. More details on implementation are described in Section 3.4.6.
- **Visibility for LOD Selection:** We utilize the number of visible pixels of geometry queried using `GL_NV_occlusion_query` in selecting the appropriate LOD. Details are discussed in Section 3.4.6.

Multiple Occlusion Tests

Our rendering algorithm performs several optimizations in order to improve the overall performance. The `GL_NV_occlusion_query` on current GPUs allows for issuing multiple occlusion queries at a time and querying the results at a later time. We traverse the scene graph in a breadth-first manner and perform all possible occlusion queries for the nodes at a given level. This traversal results in an improved performance. Note that certain nodes may be occluded at a given level and are not tested for visibility. Next we query the results and compute the visibility of each node. Let L_i be the list of nodes at level i that are being tested for visibility as well as pixel-deviation error. We generate the list L_{i+1} that would be tested at level $i+1$ by pushing the children of a node $n \in L_i$ only if its bounding box is visible and it does not satisfy the pixel-deviation-error criterion. We use an occlusion identifier for each node in the scene graph, and we exploit the parallelism available in `GL_NV_occlusion_query` by performing multiple occlusion queries at each level.

Visibility for LOD Selection

The LODs in a scene graph are associated with a screen-space projection error. We traverse the scene graph until each LOD satisfies the pixels-of-error metric. This approach, however, can be too conservative if the object is mostly occluded. We therefore utilize the visibility information in selecting an appropriate LOD or HLOD of the object.

The number of visible pixels for a bounding box of a node provides an upper bound on the number of visible pixels for its geometry. The `GL_NV_occlusion_query` also returns the number of pixels visible when the geometry is rendered. We compute the visibility of a node by rendering the bounding box of the node, and the query returns the number of visible pixels corresponding to the box. If the number of visible pixels is lower than the pixels-of-error specified by a bound, we do not traverse the scene graph any further at that node. This additional optimization is very useful if only a very small portion of the bounding box is visible and the node has a very high screen-space projection error associated with it.

3.4.7 Design Issues

Latency and reliability are two key components considered in the design of our overall rendering system. In addition to one frame of latency introduced by an occlusion switch, our algorithm introduces additional latency due to the transfer of camera parameters and visible node identifiers across the network. We also require reliable transfer of data among different GPUs to ensure the correctness of our approach.

System Latency

A key component of any parallel algorithm implemented using a cluster of PCs is the network latency introduced in transmitting the results from one PC to another during each frame. The performance of our system depends on the latency involved in receiving the camera parameters by the GPUs involved in occlusion switch. In addition, latency

is introduced in sending the camera parameters from the GPU performing HC to the GPU performing RVG. Moreover, latency is also introduced in sending the visible nodes across the network to RVG and OR. We eliminate the latency problem in receiving the camera parameters by the GPU performing HC using the switching mechanism.

Let GPU_1 and GPU_2 constitute an occlusion switch. GPU_1 performs HC for frame i and GPU_2 generates OR for frame $i + 1$. For frame $i + 1$, GPU_1 generates OR for frame $i + 2$, and GPU_2 performs HC for frame $i + 1$. Given that GPU_2 has already rendered the occluders for frame $i + 1$, it has the correct camera parameters for performing HC for frame $i + 1$. As a result, no additional latency is incurred as HC receives the camera parameters. The GPU performing OR, however, requires the camera parameters from the GPU performing RVG, and receiving these parameters introduces latency. Because HC takes some time to perform hardware cull tests before transmitting the first visible node to the GPU performing OR, this latency is usually hidden. We reduce the latency in transmitting camera parameters from HC to RVG by sending them at the beginning of each frame. Fig. 3.5 illustrates the basic protocol for transferring the camera parameters among the three GPUs. We enumerate other sources of network latency in Section 3.6.

Reliability

The correctness and conservativeness of our algorithm depend on the reliable transmission of camera parameters and the visible nodes between the GPUs. Our system is synchronized based on transmission of an end-of-frame (EOF) packet. This protocol requires us to transmit the camera parameters reliably from the GPU performing HC to the GPU performing RVG. Also, we require reliable transmission of node identifiers and EOF from the GPU performing HC to the GPUs performing OR and RVG. We have used reliable transfer protocols (TCP/IP) to transfer the data across the network.

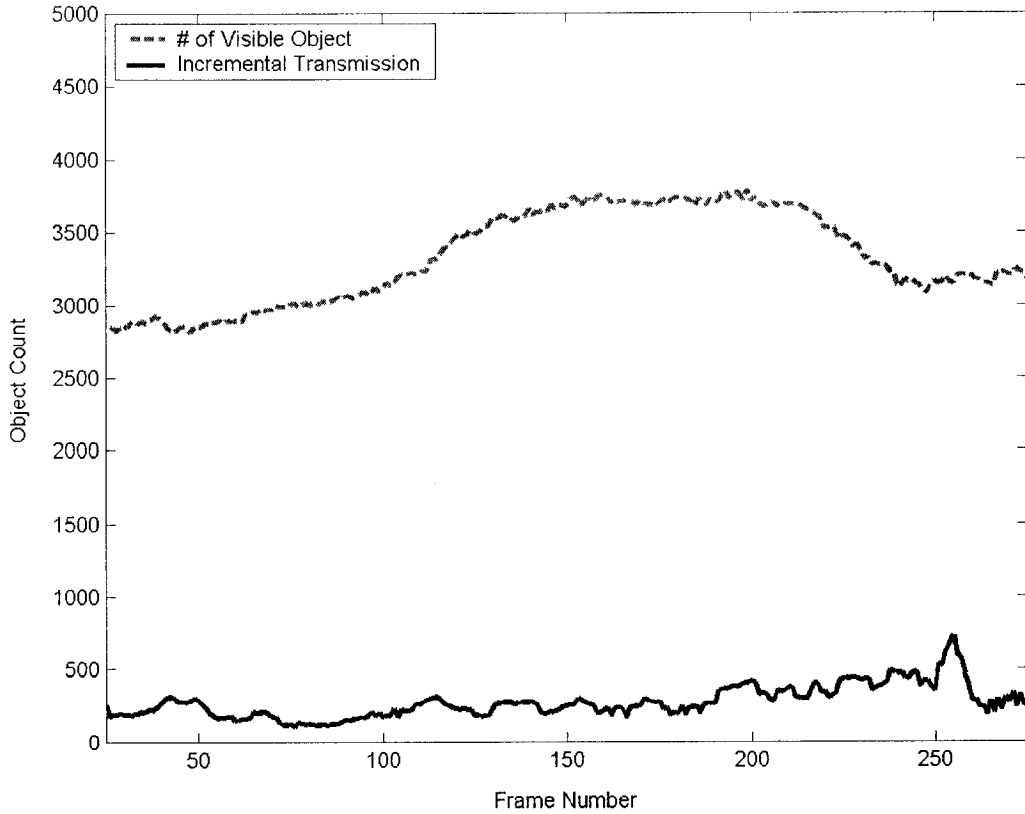


Figure 3.7: Comparison of the number of nodes transmitted with and without incremental transmission (described in Section 3.4.5) for a sample path on Double Eagle Tanker model. Using incremental transmission, we observe an average reduction of 93% in the number of nodes transmitted between the GPUs.

3.5 Implementation and Performance

In this section, we describe the implementations of our preprocessing algorithm and the run-time occlusion culling algorithm. We also compare the performance of our culling algorithm with earlier approaches.

3.5.1 Preprocessing

This section reports the values used for the preprocessing parameters mentioned in Section 3.3 and provides details about the amount of time and memory used by our preprocess.

Preprocessing Parameters

For Partition-I (Sec. 3.3.2), we have obtained good results with $t_1=1000$, $s_1 = 0.1 \times \text{model_dimension}$, $r_1=2.5$.

For Clustering (Sec. 3.3.2), we use $D = 0.1 \times \text{max_bounding_box_dimension}$. We can also work on a complete graph, but choosing a threshold in general works well and reduces the computational complexity. We choose

$V = 10^{-3} \times \text{total_scene_bounding_box_volume}$ as the maximum volume threshold. For K we use 1500, which prevents too many closely-packed objects from merging into a single cluster.

For Partition-II (Sec. 3.3.2) we use $s_2 = 2s_1$ and triangle count $t_2 = 10t_1$ but tighter bounds for the aspect ratio: $r_2 = 0.5r_1$.

Time and Space Requirements

The preprocessing step was computed on a single-processor 2 GHz Pentium 4 PC with 2 GB RAM. The preprocessing times for the Double Eagle model were 45 min for Partition-I, 90 min for Clustering, 30 min for Partition-II, 32.5 hours for out-of-core HLOD generation, and 12 min for the AABB-hierarchy generation. The size of the final HLOD scene graph representation is 7.6 GB, which is less than 2 times the original data size. The AABB-tree hierarchy occupies 7 MB space.

The main memory requirement for partitioning and clustering is bounded by the size of the largest object/cluster. For the Double Eagle it was less than 200 MB for partitioning, 1 GB for clustering, and 300 MB for out-of-core HLOD generation.

3.5.2 Run-time System

We have implemented our parallel occlusion culling algorithm on a cluster of three 2.2 GHz Pentium-4 PCs, each having 4 GB of RAM (on an Intel 860 chipset) and a GeForce 4 Ti 4600 graphics card. Each runs Linux 2.4, with bigmem option enabled giving 3.0

Model	Pixels of Error	Average FPS		
		SWITCH	Distributed GigaWalk	GigaWalk
PP	5	14.17	6.2	5.6
DE	20	10.31	4.85	3.50
B-777	15	13.01	5.82	

Table 3.1: *Average frame rates obtained by different acceleration techniques over the sample path. **FPS** = Frames Per Second, **PP** = Power Plant model, **DE** = Double Eagle Tanker model, **B-777** = Boeing 777 model*

Model	Pixels of Error	Number of Polygons		
		SWITCH	GigaWalk	Exact Visibility
PP	5	91550	119240	7500
DE	20	141630	173350	10890

Table 3.2: *Comparison of the number of polygons rendered by the two implementations to the actual number of visible polygons. **PP** = Power Plant model, **DE** = Double Eagle Tanker model*

GB of user-addressable memory. The PCs are connected via 100 Mb/s Ethernet. We typically obtain a throughput of 1-2 million triangles per second in immediate mode using triangle strips on these graphics cards. Using the NVIDIA OpenGL extension GL_NV_occlusion_query, we perform an average of around 50,000 queries per second.

The scene database is replicated on each PC. Communication of camera parameters and visible node identifiers between each pair of PCs is handled by a separate TCP/IP stream socket over Ethernet. Synchronization between the PCs is maintained by sending a sentinel node over the node sockets to mark an end of frame (EOF).

We compare the performance of the implementation of our algorithm (called SWITCH) with the following algorithms and implementations:

- **GigaWalk**: A fast parallel occlusion culling system which uses two SGI IR2 graphics pipelines and three CPUs (Baxter et al., 2002). OR and RVG are performed in parallel on two separate graphics pipelines, while occlusion culling is performed in parallel using a software-based hierarchical Z-buffer. All the interprocess communication is handled using the shared memory.

Model	Pixels of Error	Number of Objects		
		SWITCH	GigaWalk	Exact Visibility
PP	5	1557	2727	850
DE	20	3313	4036	1833

Table 3.3: Comparison of the number of objects rendered by the two implementations to the actual number of visible objects. **PP** = Power Plant model, **DE** = Double Eagle Tanker model

- **Distributed GigaWalk:** We have implemented a distributed version of GigaWalk on two PCs with NVIDIA GeForce 4 GPUs. One of the PCs serves as the occlusion server, implementing OR and occlusion culling in parallel. The other PC is used as a display client. The occlusion culling is performed in software similar to GigaWalk. Interprocess communication between PCs is based on TCP/IP stream sockets.

We compared the performance of the three systems on three complex environments: a coal fired Power Plant composed of 13 million polygons and 1200 objects, a Double Eagle Tanker composed of 82 million polygons and 127K objects, and part of a Boeing 777 composed of 20 million triangles and 52K objects. Figs. 3.8, 3.9(a), and 3.9(b) illustrate the performance of SWITCH on a complex path in the Boeing 777, Double Eagle, and Power Plant models, respectively. Notice that we are able to obtain 2-3 times speedups over earlier systems.

We have also compared the performance of the occlusion culling algorithm in terms of the number of objects and polygons rendered as compared to the number of objects and polygons exactly visible. *Exact visibility* is defined as the number of primitives actually visible up to the screen-space and depth buffer resolution from a given viewpoint. The exact visibility is computed by drawing each primitive in a different color to an “item buffer” and counting the number of colors visible. Figs. 3.10(a) and 3.10(b) show the culling performance of our algorithm on the Double Eagle Tanker model.

The average speedup in frame rate for the sample paths is shown in Table 3.1. Tables 3.2 and 3.3 summarize the comparison of the primitives rendered by SWITCH

and GigaWalk with the exact visibility for polygons and objects respectively. As the scene graph of the model is organized in terms of objects and we perform visibility tests at an object level and not at the polygon level, we observe a discrepancy in the ratios of number of primitives rendered to the exact visibility for objects and polygons.

3.5.3 Bandwidth Estimates

In our experiments, we have observed that the number of visible objects n typically ranges from about 100 to 4000 depending upon scene complexity and the viewpoint. If we render at most 30 frames per second (fps), header size h (for TCP, IP and Ethernet frame) of 50 bytes and buffer size b of 100 nodes per packet, then we require a maximum bandwidth of 8.3 MBps. Hence, our system is not limited by the available bandwidth on fast ethernet. However, the variable-window-size buffering in TCP/IP (Jacobson, 1988), introduces network latency. The incremental-transmission algorithm greatly lowers the communication overhead between different GPUs. Fig. 3.7 shows the number of node identifiers transmitted with and without incremental transmission for a sample path in the Double Eagle Tanker model. We observe a very high frame-to-frame coherence and an average reduction of 93% in the bandwidth requirements. During each frame, the GPUs need to transmit pointers to a few hundred nodes, which adds up to a few kilobytes. The overall bandwidth requirement is typically a few megabytes per second (typically less than 10MBps).

3.6 Analysis

In this section, we analyze different factors that affect the performance of occlusion-switch-based culling algorithm. One of the key issues in the design of any distributed-rendering algorithm is system latency. In our architecture, we may experience latency due to one or more of the following reasons:

1. **Network:** Network latencies mainly depend upon the implementation of transport protocol used to communicate between the PCs. The effective bandwidth varies depending on the packet size. Implementations like TCP/IP inherently buffer the data and may introduce latencies. Transmission of a large number of small packets per second can cause packet loss, and re-transmission introduces further delays. Buffering of node identifiers reduces the loss but increases network latency. Using our current implementation based on TCP/IP on the benchmarks, we have observed latencies upto 50ms per frame.
2. **Hardware Cull:** Rendering a bounding box usually requires more resources in terms of fill-rate as compared to rasterizing the original primitives. If the application is fill-limited, HC can become a bottleneck in the system. In our current implementation, we have observed that the latency in HC is less than the network latency. Using a front-based ordered culling, as described in Section 3.4.5, reduces the fill requirement involved in performing the queries and thus results in a better performance.
3. **OR and RVG:** OR and RVG can become bottlenecks when the number of visible primitives in a given frame is very high. In our current implementation, HC performs culling at the object level. As a result, the total number of polygons rendered by OR or RVG can be quite high depending upon the complexity of the model, the LOD-error threshold, and the position of the viewer. We can reduce this number by selecting a higher threshold for the LOD error.

The overall performance of the algorithm is governed by two factors: culling efficiency for occlusion culling and the overall frame rates achieved by the rendering algorithm.

- **Culling Efficiency:** Culling efficiency is measured in terms of the ratio of the number of primitives in the potentially visible set to the number of primitives visible. The culling efficiency of occlusion switch depends upon the occlusion

representation used for performing culling. A good selection of occluders is crucial to the performance of HC. The choice of bounding geometric representation used to determine the visibility of an object affects the culling efficiency of HC. In our current implementation, we have used a rectangular bounding box as the bounding volume because of its simplicity. As HC is completely GPU-based, we can use any other bounding volume (e.g. a convex polytope, k-dop) and the performance of the query will depend on the number of triangles used to represent the boundary of the bounding volume.

- **Frame Rate:** Frame rate depends on the culling efficiency, the load balancing between different GPUs, and the network latency. Higher culling efficiency results in fewer primitives rendered by OR and RVG. A good load balance between the occlusion switch and the RVG would result in maximum system throughput. The order and the rate at which occlusion tests are performed affects the load balance across the GPUs. Moreover, the network latency also affects the overall frame rate. The frame rate also varies based on the LOD selection parameter.

With faster GPUs, we would expect higher culling efficiency as well as improved frame rates.

3.6.1 Comparison with Earlier Approaches

We compare the performance of our occlusion culling algorithm with two other well-known occlusion culling algorithms: HZB (Greene et al., 1993) and HOM (Zhang et al., 1997a). Both of these approaches use a combination of object-space and image-space hierarchies and are conservative up to the image precision. The current implementations of HZB and HOM are based on frame-buffer readbacks and perform the occlusion tests in software. The software implementation incurs additional overhead in terms of hierarchy construction. Moreover, these algorithms project the object's bounding volume to the

screen-space and compute a 2-D screen-space bounding rectangle for performing the occlusion test. As a result, these approaches are more conservative as compared to occlusion-switch-based culling algorithm. Further, the depth buffer readbacks can be expensive as compared to the use of occlusion queries, especially on current PC systems. In practice, we obtained almost three times speedup over an implementation of HZB on two PCs (Distributed GigaWalk).

Our algorithm also utilizes the number-of-visible-pixels parameter returned by `GL_NV_occlusion_query` for LOD selection. This bound makes our rendering algorithm less conservative compared to earlier LOD-based rendering algorithms, which compute a screen-space bound from the object space deviation error.

3.6.2 Limitations

Occlusion-switch based culling introduces an extra frame of latency in addition to double buffering. The additional latency does not decrease the frame rate, as the second pass is performed in parallel. However, it introduces additional latency into the system; the overall algorithm is best suited for latency-tolerant applications. In addition, a distributed implementation of the algorithm may suffer from network delays, depending upon the implementation of the network-transmission protocol used. Our overall approach is general and independent of the underlying networking protocol.

Our occlusion culling algorithm also assumes high spatial coherence between successive frames. If the camera position changes significantly from one frame to the next, the visible primitives from the previous frame may not be a good approximation to the occluder set for the current frame. As a result, the culling efficiency may not be high.

Our algorithm performs culling at an object level and does not check the visibility of each triangle. As a result, its performance can vary based on how the objects are defined and represented in the scene graph.

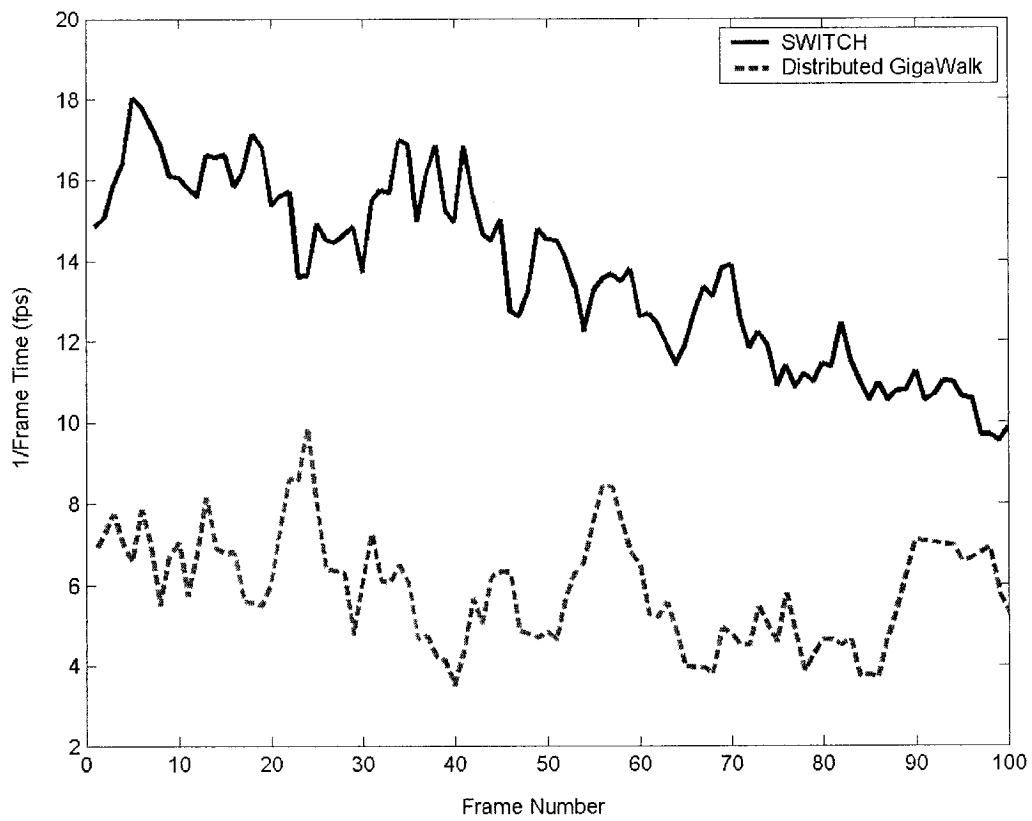
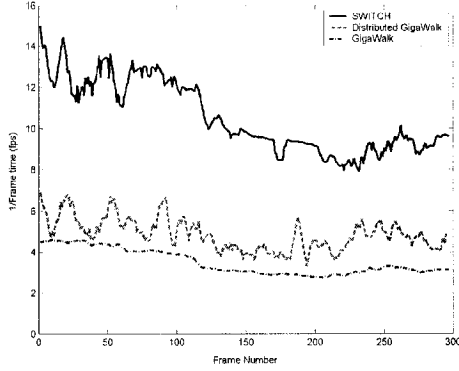
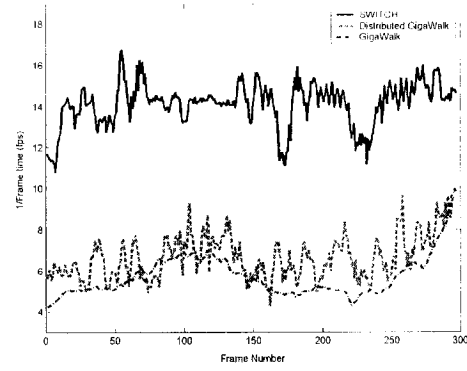


Figure 3.8: *Frame-rate comparison between SWITCH and Distributed GigaWalk at 1024×1024 screen resolution and 15 pixels of error on the Boeing model.*

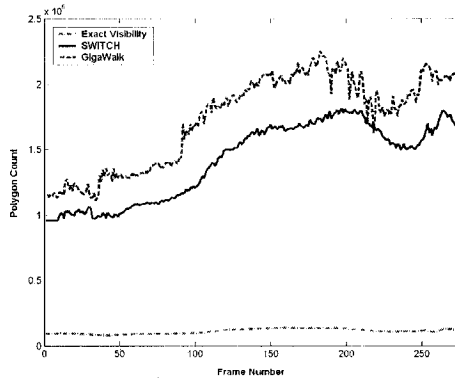


(a) Double Eagle Tanker model at 20 pixels of error

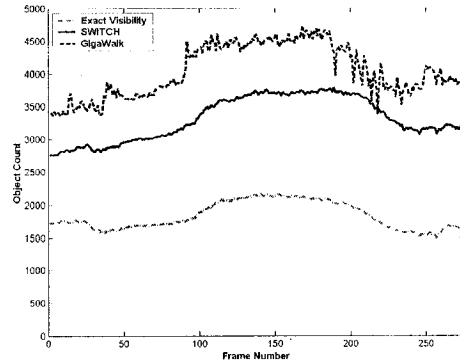


(b) Power Plant model at 5 pixels of error

Figure 3.9: *Frame-rate comparison among SWITCH, GigaWalk and Distributed GigaWalk at 1024×1024 screen resolution. We obtain 2 – 3 times improvement in the frame rate as compared to Distributed GigaWalk and GigaWalk.*



(a) At polygon level



(b) At object level

Figure 3.10: *Double Eagle Tanker: Comparison of exact-visibility computation with SWITCH and GigaWalk at 20 pixels of error at 1024×1024 screen resolution. SWITCH is able to perform more culling than GigaWalk; however, it renders one order of magnitude more triangles or twice the number of objects as compared to exact visibility.*

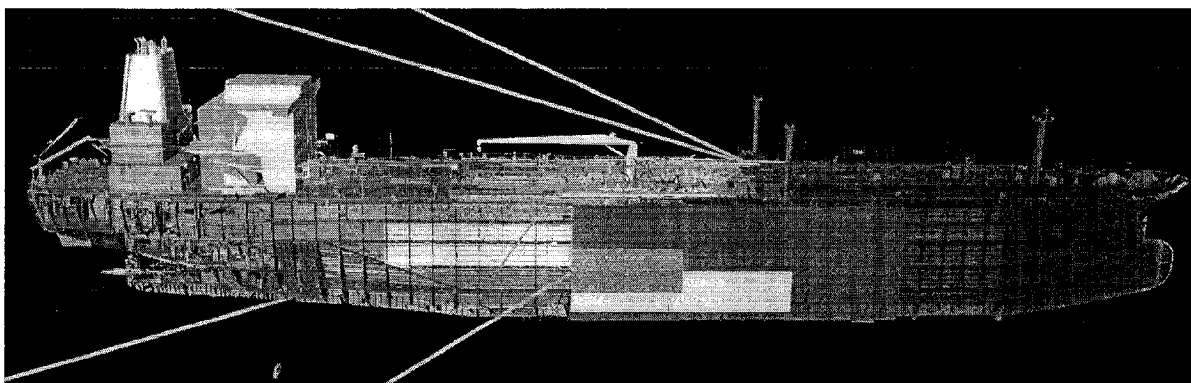
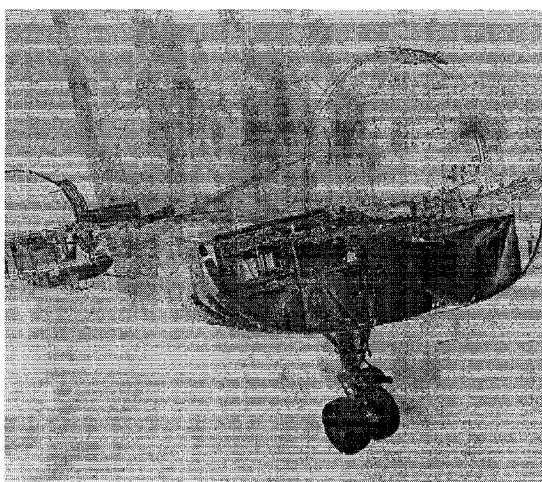
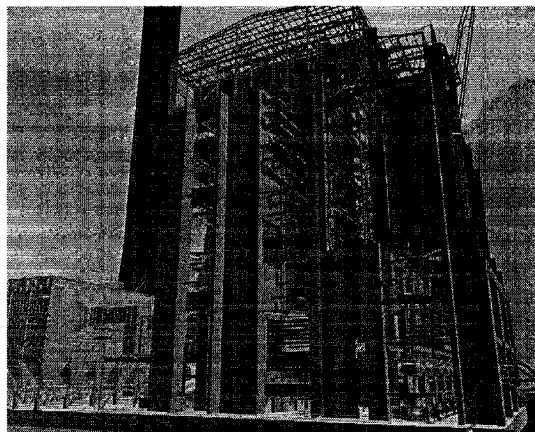


Figure 3.11: *Performance of the occlusion-switch algorithm on the Double Eagle Tanker model: This environment consists of more than 82 million triangles, and our algorithm renders it at 9 – 15 fps on a cluster of 3 PCs, each consisting of an NVIDIA GeForce 4 GPU. Occlusion switch culls away most occluded portions of the model and renders around 200K polygons in the view shown. Objects are rendered in the following colors: visible: yellow; view-frustum culled: violet; and occlusion-culled: orange.*



(a) *Portion of a Boeing 777 model rendered at 15 pixels of error. Our system, SWITCH, is able to render it at 11-18 frames per second on a 3-PC cluster.*



(b) *Power Plant model composed of more than 12.7 million triangles. SWITCH can render it at 11-19 frames per second using 5 pixels of deviation error.*

Figure 3.12: *Performance of occlusion switch on complex CAD models: Both models are rendered at 1024×1024 screen resolution using NVIDIA GeForce 4 cards.*

Chapter 4

Visibility Computations: Interactive Shadow Generation

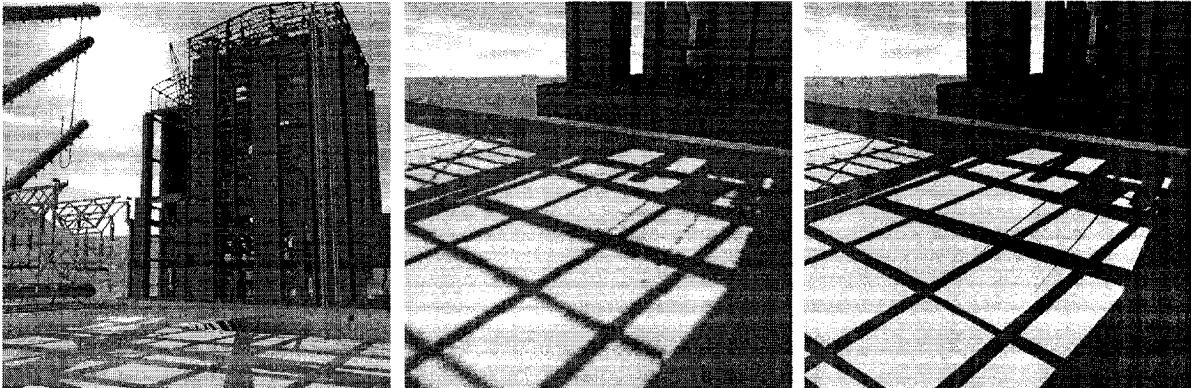


Figure 4.1: *The left image shows a snapshot generated from the application of our shadow culling algorithms and a hybrid shadow-generation technique to the Power Plant model (12.7M triangles). The middle image shows a different viewpoint generated using perspective shadow maps. Notice the aliasing artifacts. The right image highlights the sharper boundaries of the shadows generated by our interactive algorithm from the same viewpoint.*

4.1 Introduction

The generation of shadows is a classic visibility problem in computer graphics. Shadows provide important spatial cues and can greatly increase the visual realism of

computer-generated images. In this chapter, we address the problem of calculating hard-edged umbral shadows cast by a moving light source in complex static environments at interactive frame rates. Examples of these environments include architectural models, urban data sets, and CAD models of large structures such as airplanes or oil tankers. These types of scenes consist of thousands of objects, contain millions of polygons, and typically exhibit a wide depth range (as shown in Fig. 4.1).

We present a new *shadow culling* algorithm for interactive shadow generation in complex environments. The shadow culling algorithm consists of two novel components. First, we improve the techniques described in Chapter 3 for computing the potentially visible set (PVS) in complex environments using a combination of hierarchical representations, LODs, and image-space occlusion queries. The compactness of the PVS produced by this technique is necessary for object-space shadow computation. Second, we introduce a cross-culling operation involving visibility computations between the PVS computed from eye view and the PVS computed from the light view. Cross-culling results in a reduced set of potential *shadow casters* and potential *shadow receivers*. We have integrated our algorithm with a hybrid shadow-rendering method (Private Communication with Lloyd, 2003). The hybrid algorithm combines object-space shadow polygons with shadow maps. The shadow polygons are computed by a clipping algorithm using the potential shadow casters and shadow receivers identified by cross-culling. Our improved PVS-computation algorithm can also be used to accelerate other rendering applications. Likewise, our cross-culling algorithm can also be useful for other object-precision shadow algorithms such as shadow volumes (Lloyd et al., 2004).

Compared to earlier approaches, our shadow culling algorithm offers many advantages. It makes no assumptions about the input model or connectivity information; it can be used for generating sharp shadow edges, greatly reducing aliasing; and can be used with a moving light source to generate dynamic shadows at interactive rates in complex environments.

Portions of this chapter are described in (Govindaraju et al., 2003a; Lloyd et al., 2004)¹. The rest of the chapter is organized in the following manner. In Section 4.2, we describe the improved PVS-computation algorithm and discuss ways to minimize artifacts due to LODs in shadow computation. We also present the cross-culling algorithm for reducing the sizes of potential shadow casters and shadow receivers. In Section 4.3, we describe the hybrid shadow-generation algorithm and discuss the application of shadow culling algorithms to it. In Section 4.4, we describe our implementation of this hybrid algorithm and present our results. We analyze the performance of the system and discuss some of its limitations in Section 4.5.

4.2 Shadow Culling

Visibility computation is an integral part of any shadow-generation algorithm. Given a point source, the hard-edged umbral shadows can be determined by partitioning the visible surface of the eye view with respect to visibility of the light as shown in Fig. 4.2. Regions not visible to the light lie in shadow. However, exact computation of the visible surface is too slow for interactive applications and is prone to geometric-robustness problems.

In this section, we present the two components of our shadow culling algorithm:

- The PVS-computation algorithm for computing the *PVS* from the eye view (PVS_E) and the *PVS* from the light view (PVS_L). We also discuss the issues that arise from using LODs to accelerate the PVS computation.
- The cross-culling algorithm for computing the actual shadow casters and shadow receivers contributing to the shadow boundaries.

¹Joint work with Brandon Lloyd, Sung-Eui Yoon, Jeremy Wendt and Avneesh Sud

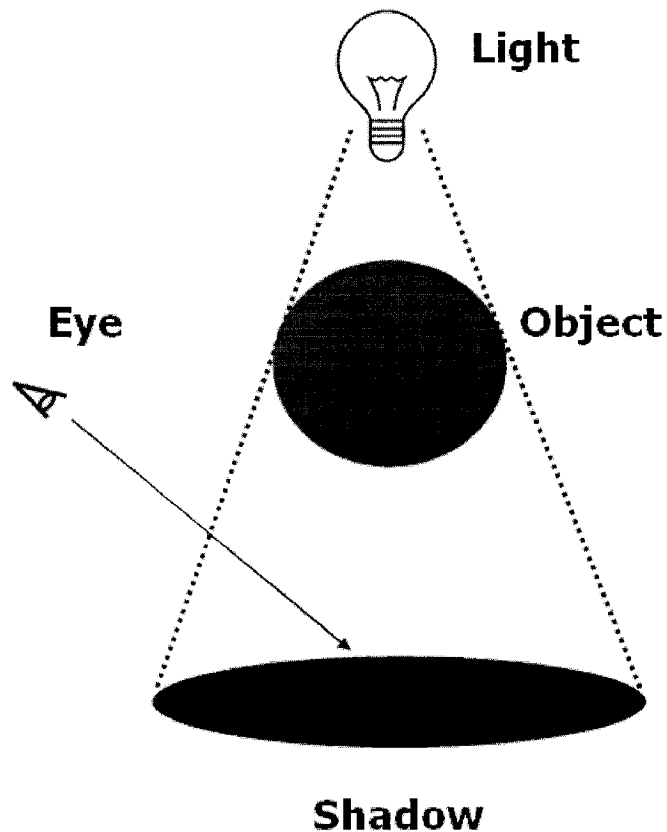


Figure 4.2: *Visibility Computation for Shadow Generation: This figure shows a simple scene with a light source placed above a sphere and a floor. Shadows are regions on the floor and sphere that are visible to the eye, but not visible to the light.*

4.2.1 LOD-based Interactive PVS Computation

Our PVS-computation algorithm is based heavily on our prior work, which combines LODs with occlusion culling. The details are discussed in Chapter 3. In chapter 3, we perform culling only at the object level. As a result, the computed PVS can be overly conservative, ranging anywhere from 200K to 450K triangles in size on our benchmark models. Although we can render a PVS of this size at interactive rates on current graphics processors, it is too large for interactive shadow-generation algorithms. We present an improved algorithm that reduces the size of PVS by almost an order of magnitude as compared to our prior algorithm and decreases the latency in the pipeline.

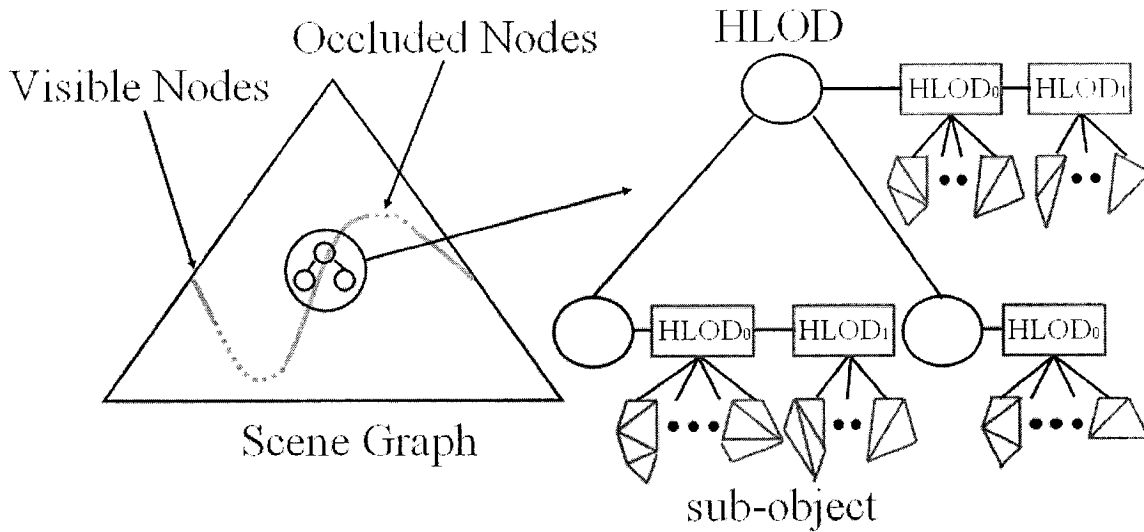


Figure 4.3: This figure shows a cut defined by the traversal of the scene graph using our improved culling algorithm. The cut is composed of visible and occluded nodes. Each node in the hierarchy is composed of several HLODs and each HLOD is further decomposed into multiple subobjects to improve the culling efficiency of our algorithm.

To improve the performance of culling, we decompose each LOD or HLOD object into a shallow hierarchy of subobjects as shown in Fig. 4.3. After object-level culling has been performed, we render the subobjects of the visible objects and use occlusion queries for checking whether they are visible. The subobject hierarchy is typically 1 - 2 levels deep. A deeper hierarchy can lead to stalls when performing image-space occlusion as explained further in Section 5.1. Each subobject is composed of k triangles, where k is typically a small number (say 1 - 10). A higher value of k reduces the number of subobjects per object, thereby reducing the number of occlusion queries to be performed. On the other hand, a lower value of k results in a much smaller PVS. If $k = 1$, the algorithm computes the smallest PVS for a given LOD-error threshold.

With a smaller PVS, fewer occluders are rendered during OR generation ². As a result, we can compute the occlusion representation and perform scene-graph culling

²Details on OR generation are discussed in Chapter 3

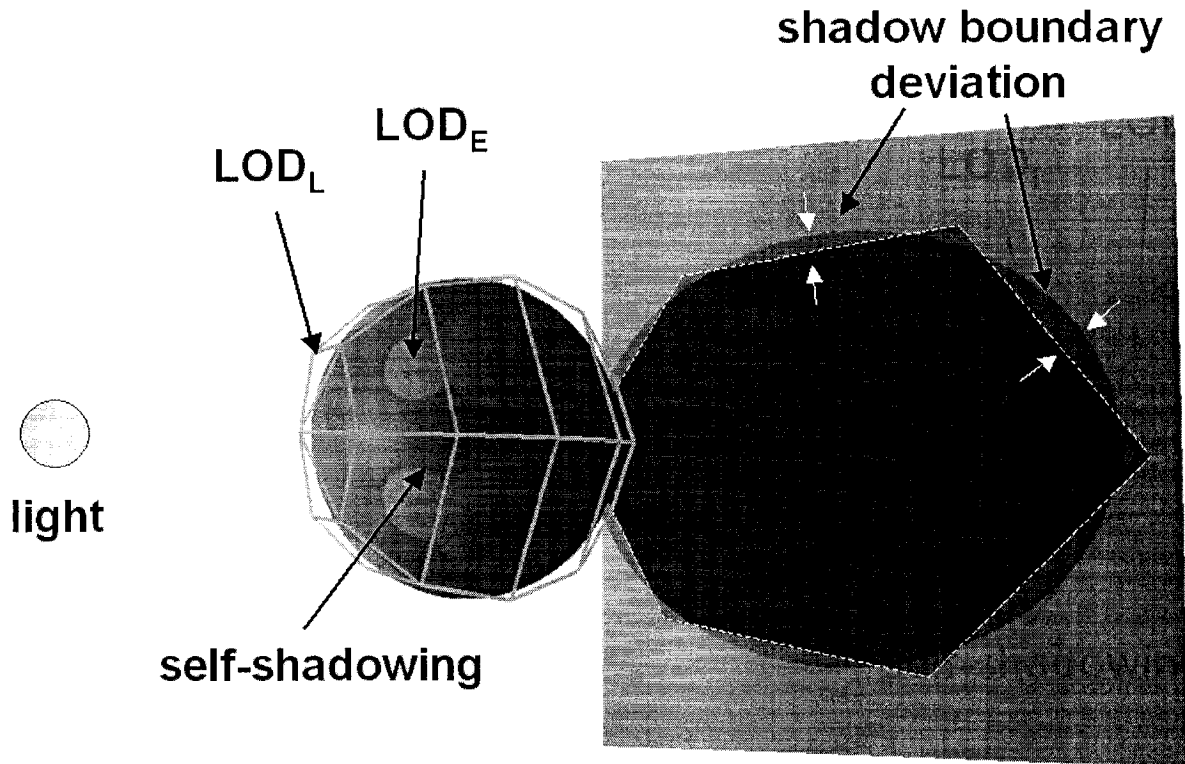


Figure 4.4: *Self-shadowing artifacts due to a naive LOD-selection algorithm. We correct this problem by using the same LOD parameter for an object when computing PVS_E and PVS_L .*

(SGC) on one GPU instead of two with little loss in overall performance. This procedure reduces the latency in the pipeline and decreases the load on the network.

Interactive Shadow Maps: The savings incurred by the improved PVS computation makes possible the rendering of large models with a shadow map on a single graphics processor. In order to render each frame, PVS_E is computed first and PVS_L second. The final image is then rendered using the depth map left over from the computation of PVS_L as the shadow map. Using this approach, we can render the 82.7 million triangle Double Eagle Tanker model with perspective shadow maps at interactive rates.

LODs and Shadow Generation

The use of LODs and HLODs in PVS computation introduces inaccuracies in shadow boundaries and can cause self-shadowing artifacts. In this section, we discuss the vari-

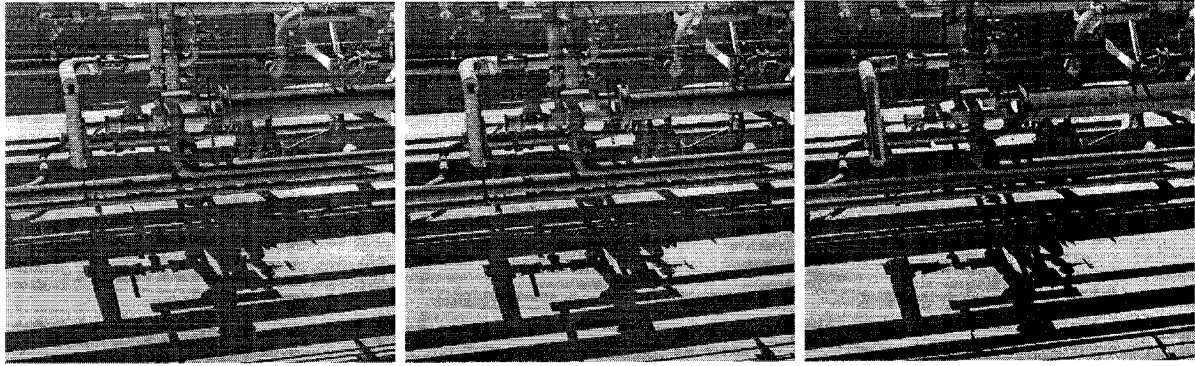


Figure 4.5: *The effect of increasing the LOD-error threshold in the Double Eagle Tanker model. These images have been generated with 0 (left), 10 (middle), and 20 (right) pixels of error.*

ous artifacts and trade-offs involved in selecting an LOD for an object when rendering shadows.

Inaccuracy in Shadow Boundaries: A shadow is formed by the projection of the silhouette of an object onto other surfaces in the scene. A deviation in the silhouette due to the use of LODs causes a deviation in the shadow boundary. This deviation is based on two factors: a distance factor that magnifies the deviation as the distance between the shadow caster and a shadow receiver is increased and an orientation factor that magnifies deviation as the orientation of the shadow receiver surface normal is almost perpendicular to the light direction (see Fig. 4.4). We cannot bound the error introduced by the orientation factor, since the orientation of the surfaces of an object can be arbitrary. But we can bound the error caused by the distance factor by using an LOD-selection metric that bounds the screen-space silhouette deviation. We use a conservative criterion based on the maximum distance between any shadow caster and shadow receiver, which is bounded by the size of the model. This ensures that, at a given point in space, the deviations in all shadow boundaries will be bounded, though the size of the error bound will be different at each point due to differences in distance and orientation with respect to the light. Fig. 4.5 shows the effect of increasing the

LOD-error threshold on shadows.

Artifacts due to LODs: In order to minimize LOD artifacts, we must select the LODs carefully. Let LOD_L and LOD_E be the LODs selected for a given object from the light view and eye view, respectively. If the object is visible in both the views, it is both a potential shadow receiver and a potential shadow caster. If LOD_L and LOD_E are not the same, self-shadowing artifacts can occur (as shown in Fig. 4.4). We propose two methods for selecting consistent LODs in both views. Both have advantages and disadvantages:

1. $\max(LOD_E, LOD_L)$: This method (Private Communication with Lloyd, 2003) produces shadows with the minimum deviation for a given LOD-error threshold, but it can result in extra geometry being rendered. Objects distant from the eye but close to the light will have a higher LOD in the eye view than is necessary to meet the LOD-error threshold. Rendering with a higher LOD than necessary should not negatively affect the final image but it does impact the performance. One potential problem for a static view is popping in the shadows, as LODs change when the light moves.
2. LOD_E : Using LOD_E all the time without regard to the position of the light can lead to shadows with large deviation in the shadow boundary. An object far from the eye will have a coarse LOD. If that object is close to the light, then inaccuracies of the LOD are magnified in the shadow. Shadows that are too coarse are usually less noticeable than objects that are too coarse, because the shadow is broken up by the geometric primitives on which it falls. This method has the advantage of keeping the number of geometric primitives rendered to a minimum. Also, the geometric primitives used for the shadows will change only if the eye view does, which usually is less distracting than if the geometric primitives change while the eye view is static.

When an object is not visible in both views, self-shadowing is not a problem because either the object is completely in shadow (not in the light view) or the shadows are not computed (not in the eye view). Even though avoiding self-shadowing artifacts is not necessary in these cases, it is still a good idea to continue to use the same LOD selection; otherwise distracting popping artifacts can occur if the LOD changes drastically when the object does become visible in both views.

Both the criteria highlighted above have relative advantages. Using either LOD-selection criterion, the LOD for an object may change when the user moves. As a result, we need to recompute PVS_L , even if the light source is static.

4.2.2 Cross-Culling

Every triangle in PVS_L is a potential shadow caster and every triangle in PVS_E is a potential shadow receiver. Cross-culling aims to reduce these sets to the triangles that actually cast or receive shadows up to image precision. PVS_E can be partitioned into three subsets with respect to PVS_L in the light view (as shown in Fig. 4.6):

- **Fully lit (\mathcal{FV}):** These triangles are fully visible in the light view and are not shadowed.
- **Fully shadowed receivers (\mathcal{SR}_F):** These triangles are totally occluded in the light view and therefore, lie fully in shadow.
- **Partially shadowed receivers (\mathcal{SR}_P):** These triangles contain shadow boundaries because they are partially occluded in the light view .

We refer to the subset of PVS_L that casts shadows on \mathcal{SR}_P as *shadow casters*, \mathcal{SC} . \mathcal{SR}_P and \mathcal{SC} can be used in both object-precision and hybrid algorithms (described in chapter 2) for computing shadow boundaries.

In order to compute \mathcal{SR}_P , we first render PVS_L to generate the depth map from the light view. Next we disable the depth mask and render the triangles in PVS_E while

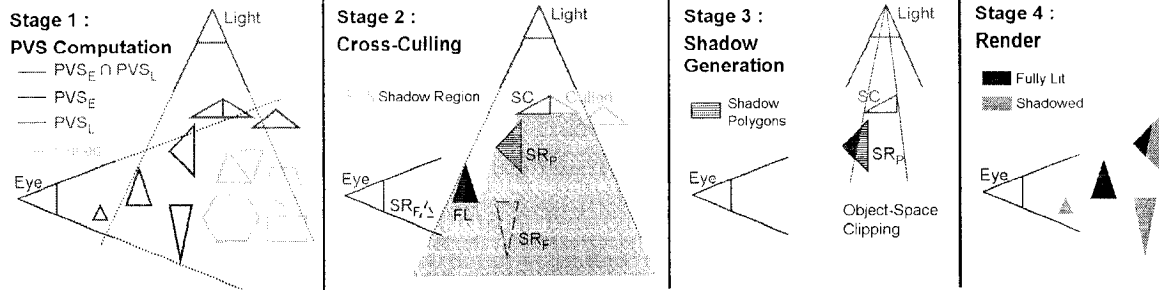


Figure 4.6: Overview of our hybrid approach showing the four stages of the algorithm and the intermediate computations.

performing occlusion queries. Occlusion queries indicate which triangles contain pixels that passed the depth test. Triangles that are completely occluded (\mathcal{SR}_F) are removed from PVS_E leaving only potentially visible triangles. These triangles are rendered with the depth function reversed so that the triangles that fail the occlusion test are actually fully visible (\mathcal{FV}). The remaining triangles are only partially visible (\mathcal{SR}_P).

In order to compute \mathcal{SC} , we need to determine the subset of PVS_L that potentially shadows \mathcal{SR}_P . This computation is performed using the stencil test. While rendering the visible triangles in the previous step, we also set the stencil where the depth test is passed, which will be in the shadowed regions of \mathcal{SR}_P . We then set the depth function to EQUAL and re-render PVS_L . The triangles that pass both the occlusion and the stencil tests are the potential shadow casters (\mathcal{SC}).

4.3 Applications

Our shadow culling algorithms can be integrated with object-precision and hybrid algorithms for generating high-quality shadows. In this section, we describe the application of our culling techniques in two recent shadow generation algorithms.

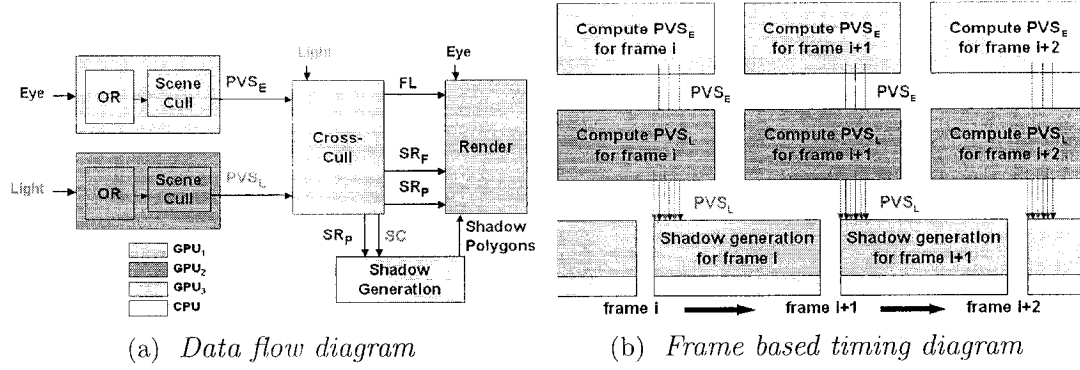


Figure 4.7: *Architecture of the Process-Parallel Algorithm: This figure shows the components of our hybrid shadow generation algorithm. Each color represents a separate graphics processor or CPU.*

4.3.1 Hybrid Shadow-Generation Algorithm

Lloyd (Private Communication with Lloyd, 2003) integrated our shadow culling algorithm with a combination of object-precision and image-precision techniques for generating high-quality shadows. The integrated algorithm proceeds in four stages (see Fig. 4.6). First, the PVS-computation algorithm described in Section 4.2 is used for computing PVS_E and PVS_L for a given eye-view and light-view. Second, cross-culling between the PVSs is used for identifying the triangles that cast and receive shadows. Third, this information is used to generate object-precision shadow polygons. Finally, the shadow polygons are combined with a shadow map in order to render the final image. A variation of the classic Atherton-Weiler-Greenberg algorithm (Atherton et al., 1978) is used for computing the shadow polygons. For more details, refer to (Govindaraju et al., 2003a).

For large models, the integrated algorithm may not be able to compute PVS_E and PVS_L , perform cross-culling, and render shadows on a single graphics processor at interactive rates. In order to increase performance, the algorithm is parallelized over three graphics cards (on three different PCs) and the computation is pipelined. The architecture of the resulting system is shown in Fig. 4.7(a). At the beginning of each frame, the PC with GPU_3 transmits the light view and the eye view cameras for the next

frame to the other GPUs. GPU_1 and GPU_2 compute PVS_E and PVS_L for the current frame in parallel (see Fig 4.7(b)). At the same time, the PC with GPU_3 receives the PVSs for the current frame, performs cross-culling, computes the shadows, and renders the final scene. This algorithm introduces one frame of latency while greatly improving the frame rate.

Network Transmission: Each PVS sent to GPU_3 comprises a list of the identifiers of the visible subobjects in the scene graph. In most cases, the change in the PVS computed between successive frames is small. By transmitting only these incremental changes we can lower network traffic among the PCs. All the communication among the PCs is synchronized using acknowledgments.

4.3.2 CC Shadow Volumes

Lloyd et al. (Lloyd et al., 2004) present an interactive shadow volume rendering algorithm for rendering complex models. The technique applies our shadow culling algorithms for removing shadow volumes that are themselves in shadow or that are not contributing to the final image. Next, a novel clamping algorithm is applied to restrict shadow volumes that actually contain shadow receivers. More details are available in (Lloyd et al., 2004).

4.4 Implementation and Performance

In this section, we describe the implementation of our shadow culling algorithms as part of a hybrid shadow-generation technique and highlight its performance on three complex environments.



Figure 4.8: *A snapshot generated from an application of our interactive shadow generation algorithm to the house model. The model has about 1.3M triangles. No LODs were used.*

4.4.1 Implementation

We have implemented our hybrid algorithm on 3 Dell Precision workstations, each with dual 1.8 GHz Pentium-IV CPUs, 2 GB of main memory and a NVIDIA GeForce-4 Ti 4600 GPU.

For increased rendering performance, we reserve 72MB of the 128MB on each GPU for storing the vertices of objects, subobjects, and bounding boxes. The memory allocated in the graphics card is sufficient to hold 6 million vertices. We perform memory management if we exceed this limit. We also use NVIDIA vertex arrays in video memory for accelerating rendering. Our algorithm keeps track of the starting location of the vertices of each object in the video memory and uses it for rendering the object's primitives.

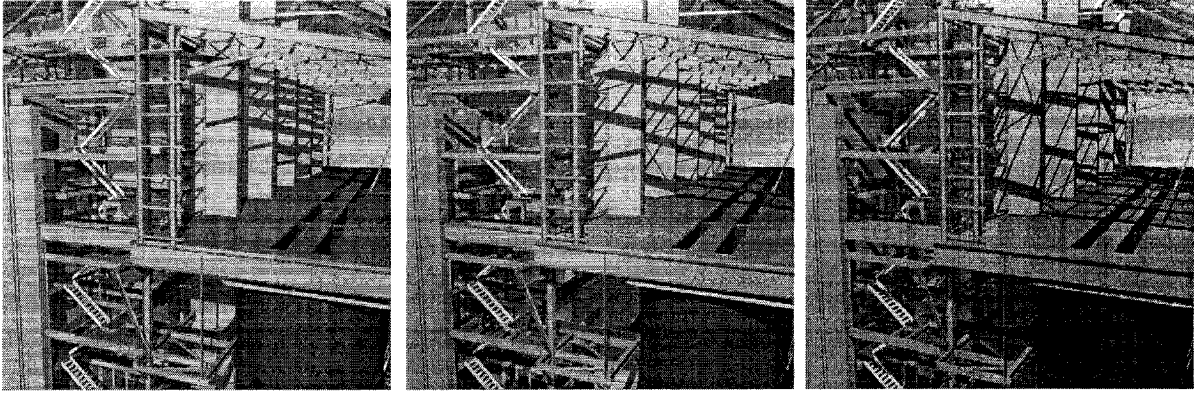


Figure 4.9: *A sequence generated by a light source moving over the Power Plant away from the viewer. Our algorithm can generate shadows at 10 frames per second on average for this model.*

We perform occlusion queries using the NVIDIA OpenGL extension *GL_NV_occlusion_query*. In order to avoid stalls in the graphics pipeline, we perform all the queries at one time and obtain the results later. In theory, current graphics processors can perform these queries at the rate of rasterization. However, we have observed considerable overhead which limits the number of queries performed. Using the current driver for NVIDIA GeForce 4 on the Linux OS, we can perform about 240K queries per second. In order to work around this limitation, we perform queries for groups of triangles, subobjects, instead of performing occlusion culling directly at the triangle level.

4.4.2 Performance

In order to measure the performance of our algorithm, we generated multiple paths for the eye and light through three large models:

- A Power Plant model (shown in Fig. 4.1) composed of more than 1,200 objects and 12.7 million triangles. We used a path that travels around the Power Plant. Fig. 4.9 shows a sequence of images generated by the moving light source in the Power Plant model.

- A Double Eagle tanker model (shown in Fig. 4.10) composed of more than 82 million triangles. We used a path generated by moving a spotlight over the top of the deck of the tanker and into the engine room.
- An architectural model (shown in Fig. 4.8) of a replicated house composed of more than 1.3 million triangles. The house contains a number of rooms with furniture. For the the path inside the house a pure shadow-map-based approach will suffer from projective aliasing.

The Power Plant and tanker models contain many long, narrow objects such as pipes, beams, and trusses. These structures are particularly problematic for shadow generation because they produce large numbers of shadow boundaries. These fine structures also tend to alias badly in shadow maps.

For the Power Plant and tanker models we used an LOD-error threshold of 10 and 20 pixels respectively, while for the house model no LODs were used because the model is not overly-tessellated. Furthermore, we used the second LOD-selection algorithm described in Section 4.2.1 (i.e., use LOD_E all the time) for generating the graphs and the paths shown in the video. We set the size of the subobjects to 8 triangles (i.e. $k = 8$). Cross-culling required around 20K occlusion queries per frame (on average) with this setting.

Fig. 4.11 shows the frame rates obtained over the different paths for each model. The average frame rate for the house model is significantly greater than that for the Power Plant or for the tanker, because the house is much smaller and relatively simple. The larger models require more time for scene-graph traversal, PVS computation, and shadow generation. The Power Plant walkthrough runs at an average of 10 frames per second while the tanker runs at an average of 7 frames per second.

The graphs in Fig. 4.12 demonstrate the performances of different culling algorithms used for shadow generation in a portion of the paths through the Power Plant and tanker. Subobject culling reduces the sizes of the PVSs computed after object culling by almost

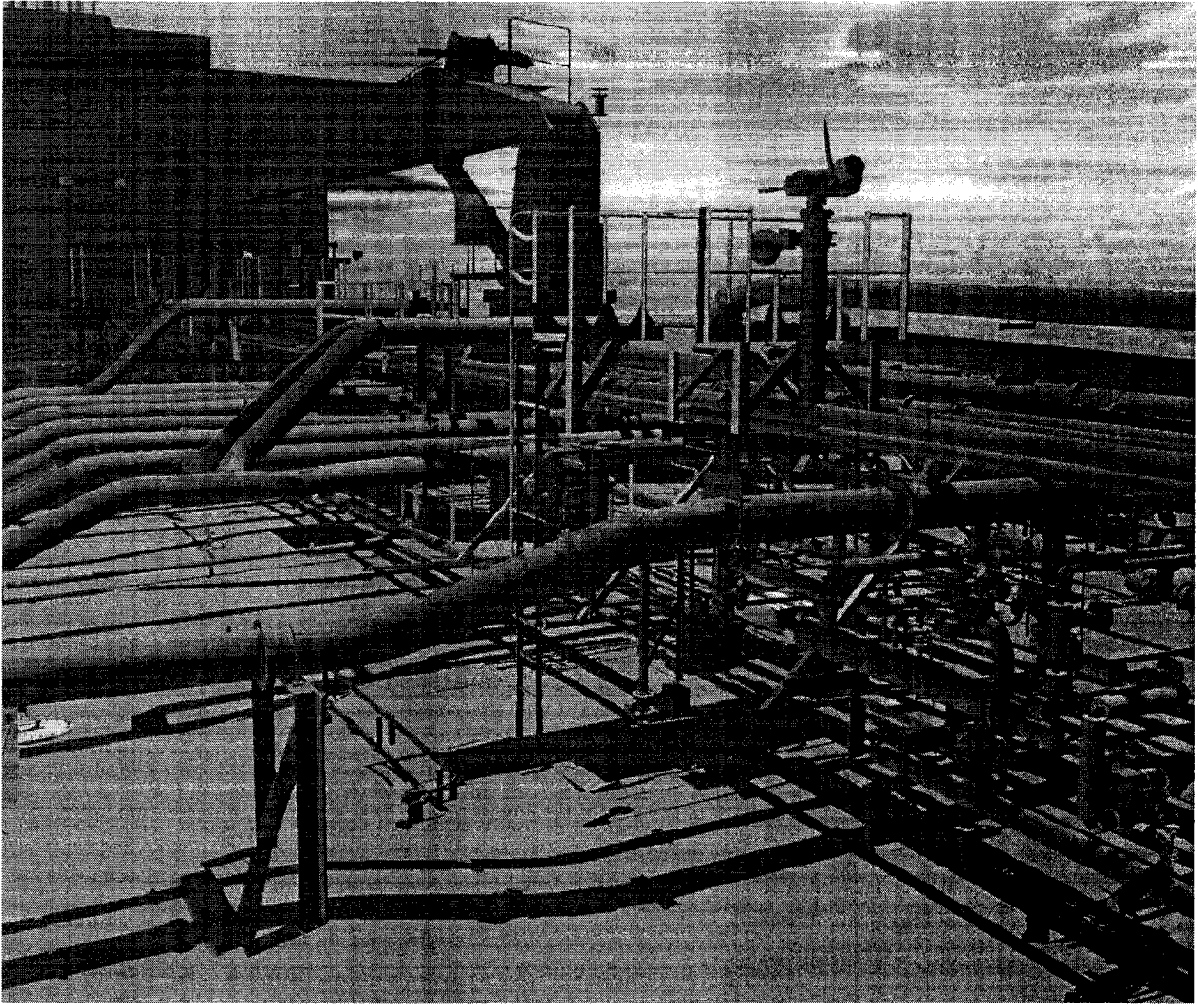


Figure 4.10: *A snapshot of the tanker model rendered using our system. The tanker has more than 82 million triangles. This view highlights the shadows generated by the long and thin pipes on the deck.*

an order of magnitude. PVS_E drops from about 100K to 11K triangles in the Power Plant and from 900K to 90K in the tanker. Cross-culling provides significant additional culling yielding sets of shadow casters and shadow receivers on the order of several thousand triangles in size, three orders of magnitude less than the model size.

4.5 Analysis and Limitations

In this section, we analyze the performance of our algorithm and discuss its limitations.

4.5.1 Interactive Performance and Load Balancing

A number of factors govern the overall performance of our algorithm, including the model complexity, the scene-graph representation, the relative positions of the light and the viewpoint, and the rate at which we can perform the occlusion queries. The frame rate is determined by the performance of each stage (as shown in Fig. 4.6) as well as by the network latency between the PCs. Network latency usually does not have much effect on performance because we exploit frame-to-frame coherence to transmit only the incremental changes in the PVSs.

The main factors that affect the system performance are the LOD-error threshold and the capabilities of the graphics processors. A higher LOD threshold improves the performance of the overall algorithm at the cost of image quality. A faster GPU will speed up both rendering and occlusion culling.

The first stage of the algorithm computes the PVS from the eye view and the light view. We compute the PVSs in parallel on separate PCs. The time it takes to compute each PVS depends on the view and the LOD-consistency criterion (described in Section 4.2.1). We found that the size of PVS_L was generally smaller as compared to the size of PVS_E and took relatively less time to compute because the light source was usually farther away from the scene than the eye. The most expensive part of the PVS computation is typically the subobject culling step because it usually requires more occlusion queries than the object-culling step.

The performance of cross-culling is directly related to the sizes of PVS_L and PVS_E . The benefit of cross-culling depends on the size of \mathcal{SR}_P relative to that of PVS_E and

the size of \mathcal{SC} relative to that of PVS_L . In general, the smaller the sizes of these sets, the larger the fraction of shadows that can be calculated with object precision, leading to higher image quality. The overhead of cross-culling is about 15-35 ms according to our benchmarks.

Load Balancing: The algorithm can spend more time in PVS computation than in the other stages. In this case we can either increase the LOD threshold in order to use coarser LODs or increase the number of triangles per subobject, k , in order to reduce the number of occlusion queries performed. If cross-culling or shadow generation becomes the bottleneck we use a smaller k . This decreases the size of PVS_E and PVS_L and the number of potential shadow casters and shadow receivers.

4.5.2 Comparison with Other Approaches

In this section, we briefly compare our algorithm with earlier approaches.

Shadow Maps: Our shadow culling algorithms can be integrated with hybrid algorithms that use shadow maps to yield higher-quality, sharper shadows than pure shadow map approaches (as shown in Fig. 4.13). Uniform shadow maps (Williams, 1978) are simple to implement but suffer from aliasing. Perspective shadow maps greatly reduce the aliasing problem for many view configurations (Stamminger and Drettakis, 2002) but cannot always eliminate it completely, especially when the field of view is narrow or when the near plane must be kept close to the viewpoint. Our algorithm can greatly reduce the aliasing artifacts present in shadow maps, while maintaining interactive frame rates.

Shadow Volumes: Shadow volumes are too slow for large models because current graphics systems cannot handle the large number of shadow casters. Currently, however, the cost of shadow volumes is dominated by the fill-rate. Recent work by Lloyd et al. (Lloyd et al., 2004) integrates our shadow culling algorithms with shadow volume clamping algorithms for reducing the fill requirement of shadow volumes and thereby,

rendering complex scenes upto a hundred thousand polygons at interactive frame rates.

Shadow Volume Reconstruction from Depth Maps: McCool [McCool 2000] uses the information in the depth buffer to construct shadow boundaries for rendering shadow volumes; thus, the shadows can be no more accurate than the information contained in the depth buffer. We use the depth buffer for computing visibility information for the objects. This information can be used for computing the shadow boundaries at object-precision (Govindaraju et al., 2003a; Lloyd et al., 2004). As a result, the quality of the shadows may be better. Moreover, McCool’s algorithm requires a depth buffer readback during each frame, which can be slow on current graphics systems (e.g. 50 milliseconds at $1K \times 1K$ resolution from a high-end PC with NVIDIA GeForce 4 card). This overhead may limit the algorithm’s usefulness for interactive performance in complex environments.

Ray Tracing: Another approach for shadow generation is ray tracing. Wald et al. [2001] described how to ray trace complex scenes at interactive rates on a cluster of commodity PCs. They were able to ray trace shadows on the Power Plant model with a resolution of 640×480 at 1 - 3 fps on a cluster of seven dual-processor PCs. The main advantages of their approach are that frame rate can scale with the number of processors and model size and that they can reduce the inaccuracies in shadow boundary by ray tracing the original geometry. However, our approach utilizes the commodity graphics processors effectively, and we are able to generate higher frame rates at a higher resolution with three graphics processors and two CPUs.

4.5.3 Limitations

Our current algorithm can only generate hard shadows from point-light sources. In theory, we can use more than one light source, but this requires additional graphics processors (one per light source) and additional computations in cross-culling and shadow generation. Our algorithm may not be suitable for latency-sensitive applications. The

PVS-computation algorithm, which is performed on separate graphics processors, introduces a latency that is typically slightly less than one frame.

Our algorithm expects high coherence between successive locations of the eye view and the light view. We assume that the set of visible primitives from the previous frame is a good approximation of the set of occluders in the current frame. If either view undergoes drastic motion, this assumption may not hold. As a result, the sizes of the PVSs, \mathcal{SR}_P , and \mathcal{SC} can become large, leading to increased frame time.

The use of LODs can introduce visual artifacts as well as inaccurate shadow boundaries. We have discussed the inaccuracies in the shadow boundaries in Section 4.2.1. Some of the popping artifacts due to the use of LODs can be eliminated by performing view-dependent simplification (Yoon et al., 2003).

Our PVS-computation and cross-culling algorithms use image-space occlusion queries to accelerate visibility computations. Very small objects or surfaces that are nearly perpendicular to the light view may be missed in rasterization. When this occurs, the visibility of the object may be misclassified resulting in missing shadows or in polygons that are incorrectly labeled as fully shadowed. Note that the effects of these artifacts will be no worse than those resulting from the use of shadow maps. Some of these problems may be avoided by increasing the resolution, by supersampling, or by performing occlusion queries in post-perspective space as in rendering perspective shadow maps. In practice, we have observed a few misclassification artifacts.

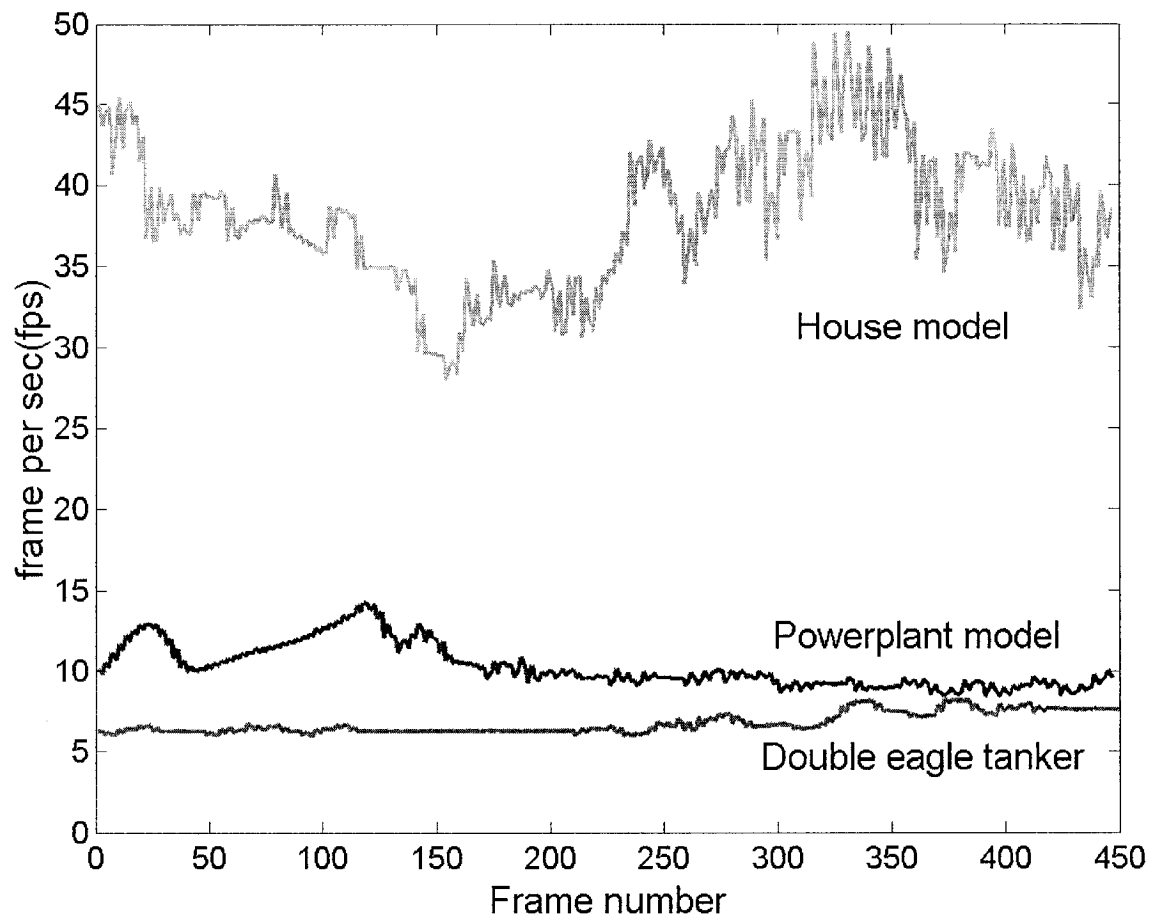


Figure 4.11: *Frame rates obtained for each model.*

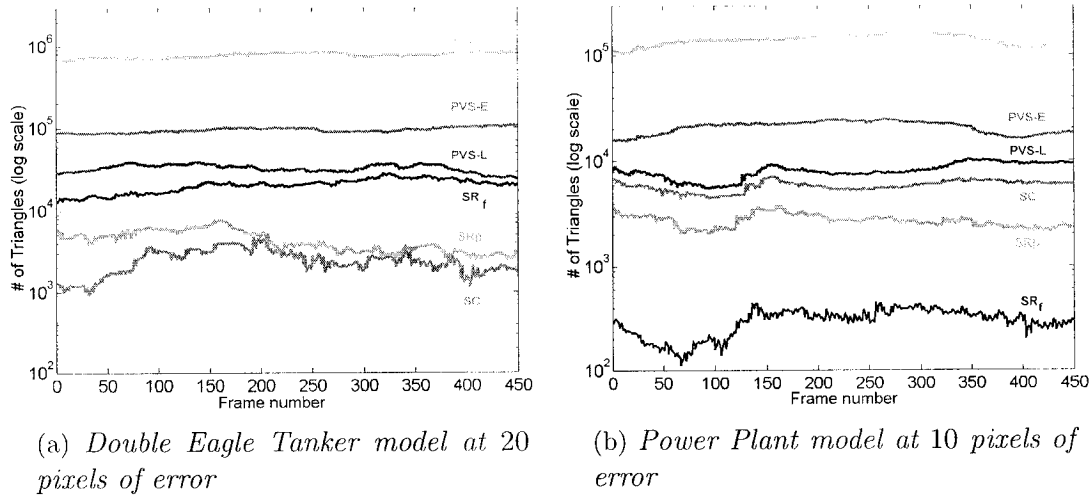


Figure 4.12: Performance of the culling techniques. OC_E refers to the number of triangles after object culling. The PVSs are obtained by performing subobject culling. There is a reduction of almost an order of magnitude in the size of PVS_E as compared to that of OC_E . After cross-culling, the sizes of the shadow casters (SC) and shadow receivers used for calculating shadow boundaries are each on the order of a few thousand.

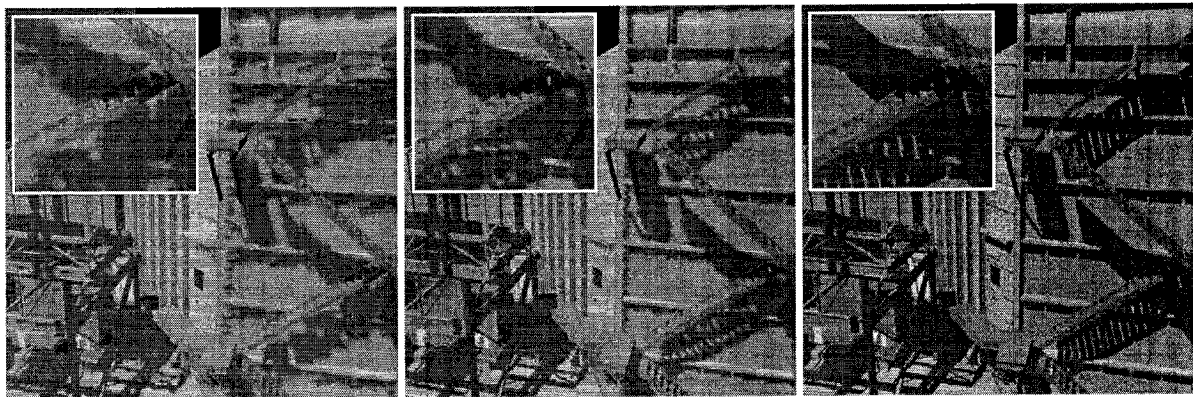


Figure 4.13: Comparison of shadows generated by uniform shadow maps (left), perspective shadow maps (middle), and our hybrid algorithm (right). Each image also includes a zoomed view of the shadow boundaries on the top left corner. Perspective shadow maps reduce some of the aliasing artifacts as compared to uniform shadow maps; however, they are unable to generate sharp shadows in many scenarios.

Chapter 5

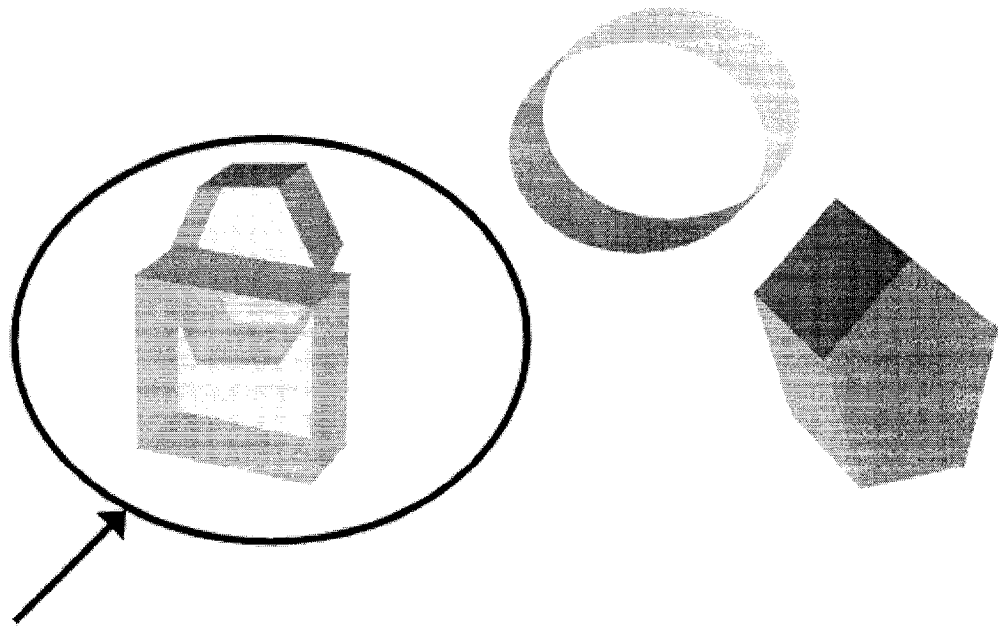
Visibility Computations: Interactive Collision Detection



Figure 5.1: *Tree with Falling Leaves:* In this scene, leaves fall from the tree and undergo nonrigid motion. They collide with other leaves and branches. The environment consists of more than 40K triangles and 150 leaves. Our GPU-based reliable collision detection algorithm, FAR, can compute all the collisions in about 35 msec per time step on a PC with 2.8 GHz Pentium IV CPU and NVIDIA GeForce FX 5950 Ultra GPU.

5.1 Introduction

Collision detection is an important problem in computer graphics, game development, virtual environments, robotics, and engineering simulations. In this chapter, we mainly address the problem of collision detection among moving objects, either rigid or deformable, using graphics processing units (GPUs).



Potentially Colliding Set

Figure 5.2: *Potentially Colliding Set: In this viewpoint, two of the four objects are in close proximity and belong to the potentially colliding set.*

We present a novel algorithm for collision or interference detection among multiple moving objects in a large environment using graphics hardware. Given an environment composed of triangulated objects, our algorithm computes a *potentially colliding set* (PCS). The PCS consists of objects that either are overlapping or are in close proximity. An example is shown in Fig. 5.2. We use visibility computations in order to prune the number of objects in the PCS. This is based on a linear-time two-pass rendering algorithm that traverses the list of objects in forward and reverse order. The visibility relationships are computed using image-space occlusion queries.

The pruning algorithm proceeds in multiple stages. Initially it computes a PCS of objects. Next it considers all *subobjects* (i.e., bounding boxes, groups of triangles, or individual triangles) of these objects and computes a PCS of the subobjects. Finally, it uses an exact collision detection algorithm in order to compute the overlapping triangles.

The complexity of the algorithm is a linear function of the input and output sizes as well as the size of the PCS after each stage. Since there are no depth-buffer readbacks, it is possible to perform the image-space occlusion queries at a higher resolution without significant degradation in performance. The additional overhead is in terms of fill rate and not the readback bandwidth.

Our pruning algorithm can also compute self-collisions in general deformable models without making assumptions about mesh connectivity. As we do not require mesh connectivity, our algorithm ignores contacts between triangles in order to avoid false collisions between connected or neighboring triangles (triangles with a shared edge or a shared vertex) and computes only penetrating triangles.

We overcome the sampling and precision problems in our pruning algorithm by “fattening” the triangles in the PCS sufficiently. We show that the *Minkowski sum* of each primitive with a sphere provides a conservative bound for performing reliable 2.5-D overlap tests using GPUs. The radius of the sphere is a function of viewport resolution and depth buffer precision. For each geometric primitive (a collection of triangles), our algorithm computes a tight bounding-offset representation. The bounding-offset representation is a union of object-oriented bounding boxes (UOBB) where each OBB encloses a single triangle. Our algorithm performs visibility queries using these UOBBs on GPUs in order to reject primitives that are not in close proximity. Overall, our algorithm guarantees that no collisions will be missed due to limited framebuffer precision or quantization errors during rasterization.

The rest of the chapter is organized as follows. In Section 5.2, we give an overview of PCS computation using visibility queries. We then present our self-collision culling and reliable-collision culling algorithms in Sections 5.3 and 5.4. We describe our overall collision detection algorithms in Section 5.5. In Section 5.6, we present our implementations and highlight the performance of our algorithms on different environments. We also analyze their accuracy and performance in Section 5.7.

5.2 Collision Detection Using Visibility Queries

In this section, we give an overview of our collision detection algorithm *CULLIDE*. We show how a PCS can be computed using image-space visibility queries, followed by exact collision detection between the primitives.

Given an environment composed of n objects, O_1, O_2, \dots, O_n . We assume that each object is represented as a collection of triangles. Our goal is to check which objects overlap and also compute the overlapping triangles in each intersecting pair. In this section, we restrict ourselves to inter-object collisions. Our algorithm makes no assumptions about the motion of objects or any coherence between successive frames. In fact, the number of objects as well as the number of triangles in each object can change between successive frames.

5.2.1 Potentially Colliding Set (PCS)

We compute a PCS of objects that either are overlapping or are in close proximity. If an object O_i does not belong to the PCS, it is implied that O_i does not collide with any object in the PCS. Based on this property, we can prune the number of object pairs that need to be checked for exact collision. This concept is similar to that of computing the potentially visible set (PVS) of primitives from a viewpoint for visibility culling (Cohen-Or et al., 2001b).

We perform visibility computations between the objects in image space to check whether they are potentially colliding or not. Given a set S of objects, we test the relative visibility of an object O with respect to S using an image-space visibility query. The query checks whether any part of O is occluded by S . It rasterizes all the objects or primitives belonging to S . O is considered *fully visible* if all the fragments generated by the rasterization of O have depth values less than those of the corresponding pixels in the frame buffer. We do not consider self-occlusion of an object (O) in checking its

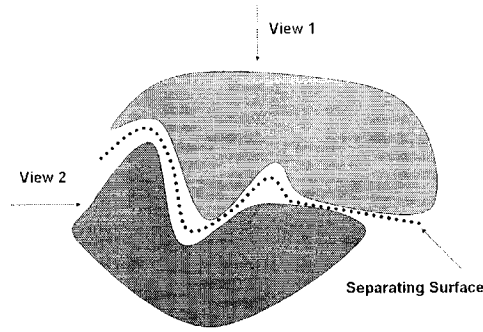


Figure 5.3: *In this figure, the two objects are not colliding. Using View 1, we determine a separating surface with unit depth complexity along the view and conclude from the existence of such a surface that the objects are not colliding. This surface's existence is a sufficient but not a necessary condition. Observe that in View 2, there does not exist a separating surface with unit depth complexity but the objects are not interfering.*

visibility status. This visibility-based pruning formulation provides a sufficient condition for the existence of a separating surface between O and S with *unit* depth complexity along the view direction. A simple two-object illustration for explaining this concept is shown in Fig. 5.3.

We use the following lemma to check whether O is overlapping with any object in S .

Lemma 1: *An object O does not collide with a set of objects S if O is fully visible with respect to S .*

Proof: The proof of this lemma is quite obvious. If O is overlapping with any object in S , then some part of O is occluded by S . We also note that this property is independent of the projection plane.

The accuracy of the algorithm is governed by the underlying precision of the visibility query. Moreover, this lemma provides only a sufficient condition and not a necessary condition for overlap between O and S .

5.2.2 Visibility-Based Pruning

We use Lemma 1 for PCS computation. Given n objects O_1, \dots, O_n , we check whether O_i potentially intersects with at least one of $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. Instead

of checking all possible pairs (which can be $O(n^2)$), we use the following lemma for designing a linear-time algorithm to compute a conservative set.

Lemma 2: *Given n objects O_1, O_2, \dots, O_n , an object O_i does not belong to the PCS if it does not intersect with $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. This test can be easily decomposed as follows: an object O_i does not belong to the PCS if it does not intersect with O_1, \dots, O_{i-1} and with O_{i+1}, \dots, O_n , $1 \leq i \leq n$.*

Proof: Follows trivially from the definition of the PCS.

We use Lemma 2 in order to determine whether an object belongs to the PCS. Our algorithm uses a two-pass rendering approach for computing the PCS. In the first pass, we check whether O_i potentially intersects with at least one of the objects O_1, \dots, O_{i-1} . In the second pass, we check whether it potentially intersects with one of O_{i+1}, \dots, O_n . If an object does not intersect in either of the two passes, then it does not belong to the PCS.

Each pass requires the object representation to be rendered twice as shown below:

1. To test whether an object representation is fully visible or not by using image-space occlusion queries. We define these queries as visibility-based overlap (VO) queries.
2. To render the object representation into the frame buffer.

We can render either all the triangles used for representing an object or a bounding box of the object. Initially, the PCS consists of all the objects in the scene. We perform these two passes in order to prune objects from the PCS. Furthermore, we repeat the process by using each coordinate axis as the axis of projection to prune the PCS further. We use Lemma 1 to check whether an object potentially intersects with a set of objects.

It should be noted that our GPU-based pruning algorithm is quite different from algorithms that prune PCS using 2-D overlap tests. Our algorithm does not perform frame-buffer readbacks and computes a PCS that is less conservative than a PCS com-

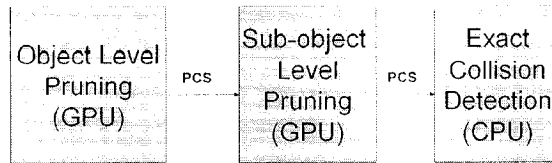


Figure 5.4: *System Architecture: The overall pipeline of the collision detection algorithm for large environments*

puted by an algorithm based on 2-D overlap tests.

5.2.3 Localizing the Overlapping Features

Many applications need to compute the exact overlapping features (e.g., triangles) for collision response. We initially compute the PCS of objects based on the algorithm highlighted above. Instead of testing each object pair in the PCS for exact overlap, we again use the visibility formulation to identify the potentially intersecting regions among the objects in the PCS. Specifically, we use a fast global pruning algorithm for localizing these regions of interest.

We decompose each object into subobjects. A subobject can be a bounding box, a group of k triangles (say a constant k), or a single triangle. We extend the approach discussed in Section 5.2.1 to the subobject level and compute the potentially intersecting regions based on the following lemma.

Lemma 3: *Given n objects O_1, O_2, \dots, O_n , with each object O_i composed of m_i subobjects $T_1^i, T_2^i, \dots, T_{m_i}^i$, a subobject T_k^i of O_i does not belong to the object's potentially intersecting region if it does not intersect with the subobjects of $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. This test can be decomposed as follows: a subobject T_k^i of O_i does not belong to the potentially intersecting region of the object if it does not intersect with the subobjects of O_1, \dots, O_{i-1} and O_{i+1}, \dots, O_n , $1 \leq i \leq n$.*

Proof: Follows trivially from Lemma 2.

In this case, we again use visibility queries for resolving the intersections among subobjects of different objects. However, we do not check an object for self-intersections or self-occlusion and therefore do not perform visibility queries among the subobjects of

the same parent object.

5.2.4 Collision Detection

Our overall algorithm CULLIDE performs pruning at two stages, object level and subobject level, and eventually checks the primitives for exact collision.

- **Object Pruning:** We perform object-level pruning by computing the PCS of the objects. We first use the AABBs of the objects in order to prune this set. Next we use the exact triangulated representation of the objects in order to prune the PCS further. If the PCS is large, we use the sweep-and-prune algorithm (Cohen et al., 1995) for computing potentially colliding pairs and decomposing the PCS into smaller subsets.
- **Subobject Pruning:** We perform subobject pruning in order to identify potential regions of each object in the PCS that may be involved in collision detection.
- **Exact Collision Detection:** We perform exact triangle-triangle intersection tests between the triangles or primitives in the PCS on the CPU in order to check whether objects collide or not.

The architecture of the overall system is shown in Fig. 5.4, where the first two stages are performed using image-space visibility queries (on the GPU) and the last stage is performed on the CPU.

5.3 Self-Collision Culling using GPUs

In this section, we present the details of our self-collision detection algorithm *S-CULLIDE* that utilizes visibility queries on GPUs for culling primitives in an object that do not self-intersect. Our algorithm extends CULLIDE for performing self-collisions among general deformable models.

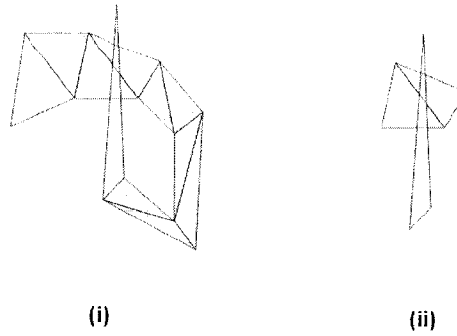


Figure 5.5: *The left image shows an object composed of triangles with shared edges and vertices. The right image shows the self-intersecting triangles in the object. Observe that these self-intersecting triangles do not share an edge or a vertex.*

Self-Collisions

Self-collision detection can be considered a special case of N-body collision detection in which each primitive in the object is tested for collision with the remaining primitives in the object. Although CULLIDE performs N-body collision detection, it cannot perform self-collision computations on general environments. In order for an algorithm to perform self-collisions in a deforming object, it has to compute collisions among the subobjects of an object. In many simulations, objects are represented as connected meshes. It is a trivial observation that a separating surface does not exist between a subobject s_i and a set of subobjects S if s_i shares one or more edges with S . The non-existence of a separating surface is due to the formulation of interference computation in CULLIDE. In CULLIDE, two objects are considered interfering if they are intersecting, and this algorithm includes the special case when these objects touch each other. Therefore two triangles sharing an edge and belonging to different subobjects are considered intersecting in CULLIDE, and so CULLIDE results in an overly conservative PCS of subobjects.

In order to overcome this problem in CULLIDE, we need to decompose objects

into subobjects that do not share edges. For general deformable objects, computing such a decomposition at run-time efficiently is difficult. We overcome the problem by modifying the collision-pruning algorithm in CULLIDE based on penetration distance between subobjects. For subobjects¹, we define the penetration distance between them as follows:

Definition 1: Given two subobjects, we define their penetration distance as the minimum translational distance for separating one subobject from the other if the subobjects are intersecting. If the subobjects are touching or not intersecting, we define their penetration distance as zero.

Computing penetration distance even for closed objects is considered difficult and expensive (Kim et al., 2002a) as the worst-case computational complexity in a scene with n polygons is $O(n^6)$. However, our self-collision pruning algorithm does not require the computation of exact penetration distance and only tests whether the penetration distance between the subobjects is zero. We define two objects as interfering in the following manner:

Definition 2: Two subobjects are considered interfering if the penetration distance computed between the subobjects is nonzero.

In the above definition, two triangles are considered interfering if they are penetrating (ignoring contacts). Using visibility computations, we can efficiently test whether the penetration distance between two subobjects is zero. A subobject s_1 is penetrating another subobject s_2 if some portion of s_1 is occluded by s_2 . We slightly modify the definition of “fully visible” status of a primitive in CULLIDE for testing whether the penetration distance of a subobject is zero.

Definition 3: Given a subobject P and a set of subobjects S , a point p_1 in S occludes

¹Penetration distance is well defined for closed objects. As subobjects are typically polygon soups that are not closed, we define penetration distance as in Def. 1.

a point p_2 in P if the distance from the viewer to p_1 is *strictly* less than the distance from the viewer to p_2 .

Definition 4: A subobject P is considered fully visible with respect to a set of subobjects S , if no point in S occludes a point in P .

Lemma 4: Given a subobject P and a set of subobjects S , if P is fully visible with respect to S along a view direction, then the penetration distance computed between P and S is zero.

Proof: Trivial.

Using Lemma 4, we design a self-collision pruning algorithm similar to the two-pass rendering approach in CULLIDE for computing self-collisions in an object. The self-collision pruning algorithm computes a compact potentially self-colliding set (PSCS) of subobjects. This set consists of the subobjects that are partially visible along the view direction. Our algorithm begins with a PSCS composed of all the subobjects in the object. We then use the object-pruning algorithm in CULLIDE with the visibility formulation in Def. 3.

5.4 Reliable Culling Using GPUs

Our algorithm CULLIDE described in Section 5.2 may miss interferences between triangles due to image-sampling and frame-buffer precision errors. In this section, we present our culling algorithm *FAR* which extends *CULLIDE* for performing reliable collision detection using graphics hardware. Our algorithm FAR aims at pruning objects in the PCS that are *not far* apart. We analyze the sampling problems caused by limited viewport resolution and present a sufficient condition for performing conservative and reliable culling.

5.4.1 Sampling Errors

We define the notation used in the rest of the section as well as the issues involved in performing interference detection on GPUs.

Orthographic projection: Let A be an axis, where $A \in \{X, Y, Z\}$ and, A_{\min} and A_{\max} define the lower and upper bounds on P_1 and P_2 along A 's direction in 3-D. Let $\text{RES}(A)$ define the resolution along an axis. The viewport resolution of a GPU is $\text{RES}(X) \times \text{RES}(Y)$ (e.g., $2^{11} \times 2^{11}$) and the depth buffer precision is $\text{RES}(Z)$ (e.g., 2^{24}).

Let O be an orthographic projection with bounds $(X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, Z_{\min}, Z_{\max})$ on the 3-D primitives. The dimension of the grid along an axis in 3-D is given by d_A where $d_A = \frac{A_{\max} - A_{\min}}{\text{RES}(A)}$. Rasterization of a primitive under orthographic projection performs linear interpolation of the vertex coordinates of each primitive and maps each point on a primitive to the 3-D grid. This mapping is based on sampling of a primitive at fixed locations in the grid. When we rasterize the primitives in order to perform VO queries, many errors arise due to sampling. There are three types of errors:

1. **Projective and perspective aliasing errors:** These errors can result in some of the primitives not getting rasterized. This error may result in an incorrect answer to the VO query.
2. **Image-sampling errors:** We can miss interferences between triangles due to sampling at the fixed locations. In this case, each triangle is sampled, but the intersection set of the triangles is not sampled (see Fig. 5.6).
3. **Depth buffer precision errors:** If the distance between two primitives is less than $\text{RES}(Z)$, the VO query may not be able to compute accurately whether one is fully visible with respect to the other.

5.4.2 Reliable VO Queries

We can overcome the errors described in Section 5.4.1 by generating a “fattened” representation T^B for each triangle T . If a triangle T interferes with a set of primitives S within a pixel, we may miss interferences because the triangle is inaccurately classified as fully visible due to the following possibilities:

- **Error 1:** a fragment is not generated during the rasterization of T or S .
- **Error 2:** a fragment is generated but does not sample the interfering points within the pixel.
- **Error 3:** a fragment is generated and samples the interfering points within a pixel, but the precision of the frame or depth buffer is not sufficient.

These errors correspond to the three types of errors discussed in Section 5.4.1. Our approach solves these problems by using “fattened” representations of triangles that serve two important functions.

- They generate at least two fragments for each pixel touched² by a triangle.
- For each pixel touched by a triangle, the depth of the corresponding two fragments bound the depth of all points of the triangle that project inside the pixel.

We use a closed, fattened representation T^B for each triangle T in CULLIDE. Doing so provides a sufficient condition for eliminating the sampling and precision errors. Suppose two primitives T_1 and T_2 intersect at some point within a pixel X that may or may not be sampled. Then T_1^B is not fully visible with respect to T_2^B , as rasterization of T_1^B generates two fragments corresponding to X and at least one of the two fragments fails the depth test. Similarly, T_2^B is not fully visible with respect to T_1^B . Therefore neither T_1 nor T_2 is pruned from the PCS. In the rest of the section, we formally prove

²A pixel is touched by a triangle if some point of the triangle projects inside the pixel.

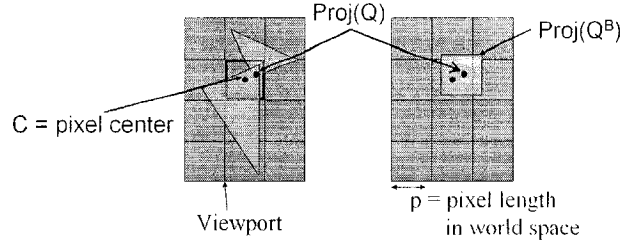


Figure 5.6: *Sampling Errors: Q is a point on the line of intersection between two triangles in 3-D. The left figure highlights the orthographic projection of Q in the screen space. The intersection of the two triangles does not contain the center of the pixel (C), and therefore, we can miss a collision between the triangles. Q^B is the Minkowski sum of Q and an axis-aligned bounding box (B) centered at the origin with dimension p . Q^B translates B to the point Q . During rasterization, the projection of Q^B samples the center of the pixel and generates at least two fragments that bound the depth of Q .*

that, for a given orthographic view, the Minkowski sum of a bounding cube B centered at the origin with T provides a conservative fattened representation T^B and eliminates sampling or precision errors irrespective of the sampling strategy used by the underlying graphics hardware. The size of the bounding cube B is a function of the world-space pixel dimensions and in practice, is very small. Therefore, P^B is a very tight fit to the original geometric primitive P .

Our algorithm does not make any assumptions about sampling the primitives within a pixel. We compute an axis-aligned bounding box B centered at the origin with dimension p where $p = \max(2*d_X, 2*d_Y, 2*d_Z)$. In practice, this bound may be conservative. If a GPU uses some uniform supersampling algorithm during rasterization, p can be further reduced. For example, if the GPU samples each pixel in the center, then p can be reduced by half.

Let B be an axis-aligned cube centered at the origin with dimension p . Given two primitives, P_1 and P_2 , let Q be a point on their line of intersection. We use the concept of the *Minkowski sum* of a primitive P with B , ($P^B = P \oplus B$), which can be defined as $\{p + b \mid p \in P, b \in B\}$. Next we show that $P \oplus B$ can be used to perform reliable VO queries. We first state two lemmas and use them to derive the main result as a theorem.

Lemma 5: Under orthographic transformation O , the rasterization of the Minkowski sum $Q^B = (Q \oplus B)$, where Q is a point in 3D space that projects inside a pixel X ,

samples X with at least two fragments bounding the depth value of Q .

Proof: Q^B is a box centered at Q , and its projection covers the center of X , as shown in Figure 5.6. As a result, Q is sampled by the rasterization hardware, and two fragments that bound the depth of Q are generated. \square

Lemma 6: Given a primitive P_1 and its Minkowski sum $P_1^B = P_1 \oplus B$. Let X be a pixel partly or fully covered by the orthographic projection of P_1 . Let us define MIN-DEPTH(P_1, X) and MAX-DEPTH(P_1, X) as the minimum and maximum depth values, respectively, of the points of P_1 that project inside X , respectively. The rasterization of P_1^B generates at least two fragments whose depth values bound both MIN-DEPTH(P_1, X) and MAX-DEPTH(P_1, X) for each pixel X .

Proof: The proof follows from Lemma 5. This lemma indicates that at least two fragments are generated after rasterizing P_1^B such that their depth values provide lower and upper bounds to the depth of all points of P_1 that project inside X . This result holds irrespective of projective or perspective errors. \square

Theorem 1: Given the Minkowski sum of two primitives with B , P_1^B and P_2^B . If P_1 and P_2 intersect, then a rasterization of their Minkowski sums under orthographic projection overlaps in the viewport.

Proof: Let P_1 and P_2 intersect at a point Q inside a pixel X . Based on Lemma 2, we can generate at least two fragments rasterizing P_1^B and P_2^B . These fragments bound all the 3-D points of P_1 and P_2 that project inside X . Showing that the pairs (MIN-DEPTH(P_1, X), MAX-DEPTH(P_1, X)) and (MIN-DEPTH(P_2, X), MAX-DEPTH(P_2, X)) overlap is sufficient. This observation follows trivially as MIN-DEPTH(P_1, X) \leq Depth(Q), MIN-DEPTH(P_2, X) \leq Depth(Q) and MAX(P_1, X) \geq Depth(Q), MAX(P_2, X) \geq Depth(Q). \square

5.4.3 Collision culling

A corollary of Theorem 1 is that if P_1^B and P_2^B do not overlap, then P_1 and P_2 do not overlap. In practice, this test can be conservative, but it will not miss any collisions

because of the viewport or depth resolution. However, the Minkowski sums, P_1^B and P_2^B , are only useful when the primitives are projected along the Z-axis. In order to generate a view-independent bound, we compute the Minkowski sum of a primitive P with a sphere S of radius $\sqrt{3}p/2$ centered at the origin. The Minkowski sum of a primitive with a sphere is the same as the *offset* of that primitive.

5.5 Interactive Collision Detection

In this section, we present our overall collision detection algorithms – CULLIDE, S-CULLIDE, and FAR – for computing all the contacts among multiple moving objects in a large environment. These algorithms use the visibility pruning algorithm described in Section 5.2.2. The algorithms are general and applicable to all environments. We also highlight many optimizations and the visibility queries used for accelerating the performance of our algorithms.

5.5.1 Pruning Algorithm

For performing linear-time PCS pruning, we use a two-pass rendering algorithm based on the visibility formulation defined in Section 5.2.2. In particular, we use Lemma 2 for computing the PCS. In the first pass, we clear the depth buffer and render the objects in the order O_1, \dots, O_n along with image-space occlusion queries. In other words, for each object in O_1, \dots, O_n , we render the object and test if it is fully visible with respect to the objects rendered prior to it. In the second pass, we clear the depth buffer and render the objects in the reverse order O_n, O_{n-1}, \dots, O_1 along with image-space occlusion queries. We perform the same operations as in the first pass while rendering each object. At the end of each pass, we test whether an object is fully visible. An object classified as fully visible in both the passes does not belong to the PCS.

5.5.2 Visibility Queries

Our visibility-based PCS-computation algorithm is based on hardware visibility query which determines if a primitive is *fully visible* or not. Current commodity graphics hardware supports an image-space occlusion query that checks whether a primitive is visible. These queries scan-convert the specified primitives and determine whether the depth of any pixel changes. Various implementations are provided by different hardware vendors, and each implementation varies in its performance as well as functionality. Some of the well-known occlusion queries based on the OpenGL extensions include the GL_HP_occlusion_test (http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt) and the NVIDIA OpenGL extension GL_NV_occlusion_query (http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt). The GL_HP_occlusion_test returns a boolean answer after checking whether any incoming fragment passed the depth test, whereas the GL_NV_occlusion_query returns the number of incoming fragments that passed the depth test.

We need a query that tests whether all the incoming fragments of a primitive have depth values *less* than the depth values of the corresponding fragments in the frame buffer. In order to support such a query, we change the depth test to pass only if the depth of the incoming fragment is greater than or equal to the depth of the corresponding fragment in the frame buffer. With this depth-comparison function, we use an image-space occlusion query for testing whether a primitive is not visible when rendered against the depth buffer. If the primitive is classified as not visible, then each incoming fragment has a depth value less than the corresponding depth value in the frame buffer, thus providing a visibility query for testing whether a primitive is fully visible. Note that we need to disable the depth writes so that the change in the depth function does not affect the depth buffer. We refer to these queries as *full-visibility* queries in the rest of the chapter. These queries can sometimes stall the graphics pipeline while waiting for

results. In Section 5.5.7, we describe techniques to avoid these stalls.

Bandwidth Requirements: Occlusion queries can be performed at the rate of the rasterization hardware and involve very low bandwidth requirements in comparison to frame-buffer readbacks. If we perform n occlusion queries, we readback n integers for a total of $4n$ bytes sent to the host CPU from the GPU. Moreover, the bandwidth requirement for n occlusion queries is independent of the resolution of the frame buffer.

5.5.3 Multiple-Level Pruning

We extend the visibility pruning algorithm to subobjects in order to identify the potentially intersecting regions among the objects in the PCS. We use Lemma 3 for performing subobject level pruning. We render each subobject for every object in the PCS with a full-visibility query. The subobject could be a bounding box, a group of triangles, or even a single triangle.

The following is the pseudocode of the algorithm.

- **First pass:**

1. Clear the depth buffer (use orthographic projection).
2. For each object O_i , $i = 1, \dots, n$
 - Disable the depth mask, and set the depth function to GL_GEQUAL.
 - For each subobject T_k^i in O_i
 - Render T_k^i using an occlusion query.
 - Enable the depth mask, and set the depth function to GL_LEQUAL.
 - For each subobject T_k^i in O_i

Render T_k^i .

3. For each object O_i , $i = 1, \dots, n$
 - For each subobject T_k^i in O_i

Test whether T_k^i is not visible with respect to the depth buffer. If it is not visible, set a tag in order to note it as fully visible.

- **Second pass:**

Same as first pass, except that the two “For each object” loops are run with $i = n, \dots, 1$ and we perform occlusion queries only if the primitive is fully visible in the first pass.

5.5.4 CULLIDE

CULLIDE performs object-level pruning, subobject-level pruning, and triangle intersection tests among the objects in PCS.

Object-Level Pruning

We perform object-level pruning in order to compute the PCS of objects. Initially, all the objects belong to the PCS. We first perform pruning along each coordinate axis, using the axis-aligned bounding boxes as the object’s representation for collision detection. The pruning is performed until the PCS does not change between successive iterations. We also use the object’s triangulated representation for further pruning the PCS. The size of the resulting set is expected to be small, and we use all-pair bounding-box overlap tests for computing the potentially intersecting pairs. If the size of this set is large, then we use the sweep-and-prune technique (Cohen et al., 1995) to prune this set instead of performing all-pair tests.

Subobject-Level pruning

We perform multiple-level pruning in order to identify the potentially intersecting triangles among the objects in the PCS. We group adjacent local triangles (say k triangles) to form a subobject used in multilevel pruning and prune the potential regions

considerably. This technique improves the performance of the overall algorithm because performing a fully visible query for each single triangle in the PCS of objects can be expensive. At the next level, we consider the PCS of subobjects and perform pruning using each triangle as a subobject. The multiple-level subobject pruning is performed across each axis.

Intersection Tests

We perform exact collision detection between the objects involved in the potentially colliding pairs by testing their potentially intersecting triangles.

5.5.5 S-CULLIDE

Our self-collision-detection algorithm is very similar to our pruning algorithm described in Section 5.5.3. We use the two-pass pruning algorithm on the primitives in the PSCS and determine a smaller PSCS. We then perform triangle intersection tests on the CPU in order to determine the pairs of overlapping triangles in the PSCS.

Although the pruning algorithm in S-CULLIDE is similar to that in CULLIDE, there are two key differences between them:

- **Image-space queries:** CULLIDE describes an implementation where the visibility query determines whether all the incoming fragments have depth values *less than* the depth values of corresponding fragments in the frame buffer. Our implementation requires a visibility query that determines whether the depth values of all the incoming fragments is *less than or equal* to the depth values of corresponding fragments in depth buffer. However, current graphics hardware does not support such queries. In order to support such a query, we change the depth test to pass an incoming fragment if its depth is *greater than* the depth of the corresponding fragment in the depth buffer. As mentioned in Section 5.5.2, we disable depth writes while performing these queries.

- **Object representations:** The object representations used in CULLIDE can be a group of triangles or any bounding-volume representation of the object. However, our pruning algorithm would be too conservative if we used bounding-volume representations. In fact, our algorithm requires the use of original geometric primitives for defining subobjects.

5.5.6 FAR

In this section, we present our reliable collision culling algorithm. We first describe the computation of a bounding-offset representation for each primitive and integrate it with CULLIDE for interactive collision detection.

Bounding-Offset Representations

In order to overcome sampling errors, we use a bounding offset for each primitive as implied by Theorem I. Our collision culling algorithm renders bounding-offset representations to cull away primitives that are not in close proximity. Several choices are possible for computing bounding offsets, and they either trade off tightness of fit for a lower rendering cost.

- **Exact Offsets:** The boundary of an exact offset of a triangle consists of piecewise linear and spherical surfaces. In particular, the Minkowski sum of a triangle T and a sphere S centered at the origin is the union of three edge-aligned cylinders of thickness $Radius(S)$, three spheres S centered at the vertices, and two triangles. The two triangles are shifted along the normal of the original triangle by the $Radius(S)$. The exact offset is the tightest fitting volume that can be rendered using graphics processors ensuring reliable interference computation. Using fragment programs, it is possible to render the exact-offset representation for each triangle but can be relatively expensive.

- **Bounded Exact Offsets:** Another possibility is to bound the exact offset of triangles tightly using three edge-axis-aligned bounding boxes, each bounding a cylinder and a sphere. This representation is a tighter fit and replaces each triangle with three bounding boxes and two triangles, thus generating 30 vertices. In our implementation, we observed that the tight fit provides better culling but is vertex-transform limited.
- **Union of Object-Oriented Bounding Boxes (UOBBS):** A tight-fitting conservative bounding representation for a primitive is a union of the object-oriented bounding boxes (OBBs) where each OBB encloses a single triangle of the primitive. Given a triangle T , we compute the tightest fitting rectangle R that encloses T ; one of its axes is aligned with the longest edge of the triangle. We compute the OBB for a triangle as the Minkowski sum of B and R , where B is a locally axis-aligned bounding cube of width $Diameter(S)$. The width of the OBB, along a dimension orthogonal to the plane containing R , is set equal to $\sqrt{3}p$. The bounding offset of a triangulated object is the union of OBBs of each triangle (see Fig. 5.7). We render this bounding offset by rendering each OBB separately and performing VO queries. In practice, this is a very tight bounding volume for an object, as compared to using a single sphere, an AABB (axis-aligned bounding box) or an OBB that encloses the entire object.

Our algorithm uses UOBBS as bounding-offset representations as shown in Fig. 5.7 for reliable collision culling. The computation of an OBB for a triangle requires 24 multiplications, 41 additions, 6 divisions, and 2 comparison operations. Further optimizations such as shared edges between adjacent triangles can be used to reduce the number of operations. Alternatively, other tight representations such as triangular prisms could be used, but these can be expensive to compute and are more conservative for long, skinny triangles.

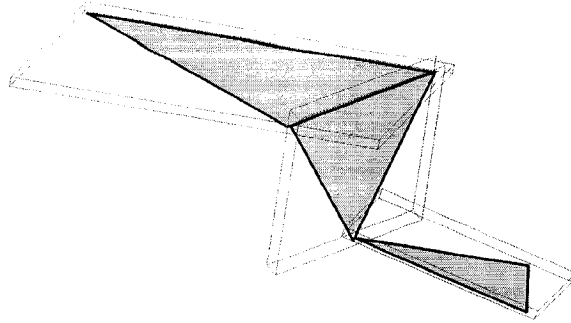


Figure 5.7: *This image shows an object with three triangles and its bounding-offset representation (UOBB) in wireframe. The UOBB is represented as the union of the OBBs of each triangle. In practice, this bounding offset is a tight-fitting bounding volume and is used for culling.*

Algorithm

We have integrated our reliable culling algorithm with CULLIDE in order to perform reliable collision detection between objects in a complex environment. As described in Section 5.2, CULLIDE uses VO queries to perform collision culling on GPUs. We extend CULLIDE to perform reliable collision culling on GPUs by using the reliable VO queries described in Section 5.4.2. For each primitive in the PCS, we compute its bounding-offset (i.e., union-of-the-OBBs) representation and use the bounding-offset representations in CULLIDE for testing whether the primitives belong to the PCS.

Our collision detection algorithm, FAR, proceeds in three steps:

1. Compute the PCS at the object level. We use sweep-and-prune (Cohen et al., 1995) on the PCS for computing the overlapping pairs at the object level.
2. Compute the PCS at the subobject level as well as the overlapping pairs.
3. Perform exact interference tests between the triangles on the CPU (Moller, 1997).

Fill Reduction: Bounding-offset representations generate nearly twice the amount of fill in comparison to the original geometric primitives. As the offset representation

for each triangle is closed, we can reduce the fill requirements for our algorithm by a factor of two by using face culling. In our optimized algorithm, we cull front faces while rendering the offset representations with occlusion queries, and we cull back faces while rendering the offset representations to the frame-buffer. These operations can be performed efficiently using *back-face culling* on graphics hardware.

Localized Distance Culling

Many algorithms aim to compute all pairs of objects whose separation distance is less than a constant distance D . In this case, we modify FAR so that it culls away primitives whose separation distance is more than D . Given a distance D , our goal is to prune triangles further away than D . We can easily modify the reliable culling algorithm presented above to perform this query. We compute the offset of each primitive by using a sphere of radius $\frac{D}{2} + \frac{\sqrt{3}p}{2}$, rasterize these offsets, and prune away a subset of primitives whose separation distance is more than D .

Accuracy

We perform reliable VO queries by rendering the bounding offsets of primitives. Theorem I guarantees that we will not miss any collisions due to the viewport resolution or sampling errors. We perform orthographic projections as opposed to perspective projections. Further, the rasterization of a primitive involves linear interpolation along all the dimensions. As a result, the rasterization of the bounding offsets guarantees that we will not miss any collision due to depth-buffer precision. If the distance between two primitives is less than the depth buffer precision, $\frac{1}{\text{RES}(Z)}$, then the VO query on their offsets will always return them as overlapping. Consequently, the accuracy of the reliable culling algorithm is governed by the accuracy of the hardware used for performing vertex transformations and mapping to the 3-D grid. For example, many of the current GPUs use IEEE 32-bit floating-point hardware for performing these computations.

5.5.7 Optimizations

In this section, we highlight a number of optimizations used for improving the performance of our algorithms.

- **AABBs and Orthographic Projections:** We use orthographic projection of axis-aligned bounding boxes. These could potentially provide an improvement factor of six in the rendering performance. Orthographic projection is used because of its speed and simplicity. In addition, we use axis-aligned bounding boxes for pruning the objects for intersection tests.
- **Visibility Query Returning Z-fail:** A hardware visibility query providing z-fail (in particular, a query for testing whether z-fail is non-zero) reduces the amount of rendering by a factor of two for AABBs under orthographic projections. This query allows us to update the depth buffer along with the occlusion query, thus improving performance by a factor of two. We take additional care in terms of ordering the view-axis-perpendicular faces of the bounding boxes and ensure that the results are not affected by possible self-occlusion, thus not affecting the query result by self-occlusion.
- **Avoiding Stalls:** We utilize `GL_NV_occlusion_query` in order to avoid stalls in the graphics pipeline. We query the results of the occlusion tests at the end of each pass, improving the performance of our algorithm by a factor of four when compared to the performance of a system using `GL_HP_occlusion_test`.
- **Rendering Acceleration:** We use vertex arrays in video memory in order to improve the rendering performance by copying the object representation to the video memory. The rendering performance can be further improved by representing the objects in terms of triangle strips and using them along with vertex arrays.

5.6 Implementation

In this section, we present the implementation details of our algorithms and describe the benchmarks used for measuring their performance.

5.6.1 CULLIDE

We have implemented CULLIDE on a Dell Precision Workstation consisting of a 2.4 GHz Pentium IV CPU and a GeForce FX Ultra 5800 GPU. We are able to perform around $400K$ image-space occlusion queries per second on this card. We have tested our system on four complex simulated environments.

- **Environment 1:** It consists of 100 deformable, open-ended cylinders moving in a unit cube with a density of 5 - 6%. Each object consists of 200 triangles. The average collision-pruning time is around 4 ms at an image-space resolution of 800×800 . A snapshot from this environment is shown in Fig. 5.15(a).
- **Environment 2:** It consists of six deformable tori, each composed of 20,000 triangles. The scene has an estimated density of 6 - 8%. The average collision-pruning query time is around 8 ms. A snapshot from this environment is shown in Fig. 5.15(b).
- **Environment 3:** It consists of two highly tessellated models: a bunny (35K triangles) and a dragon (250K triangles). In Fig. 5.15(c), we show a relative configuration of the two models, with different colors used for highlighting the triangles that belong to the PCS. A zoomed-in view of the intersection region is shown in Fig. 5.15(d). Performing each collision query takes about 40 ms.
- **Breaking Objects:** We used our collision detection algorithm to generate a real-time simulation of breaking objects. Fig. 5.16 highlights a sequence from our dynamic simulation in which the bunny and the dragon collide and the dragon

breaks into multiple pieces due to the impact. The total number of objects and the number of triangles in each piece are changing over the course of the simulation. Earlier collision detection algorithms are unable to handle such scenarios in real-time. CULLIDE takes about 35 ms (on average) to compute all the overlapping triangles during each frame.

5.6.2 S-CULLIDE

We have implemented S-CULLIDE on a Dell Precision workstation with a 2.8 GHz Xeon processor, 1 GB of main memory, and an NVIDIA GeForce FX 5950 Ultra graphics card. We used GL_NV_occlusion_query for performing the visibility queries at a viewport resolution of 1400×1400 .

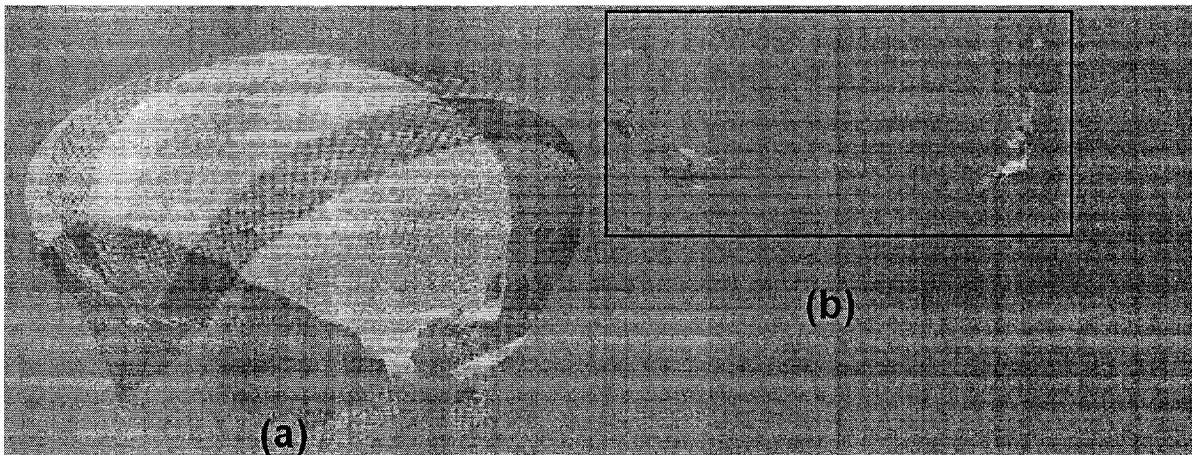


Figure 5.8: *Results of our self-collision algorithm on a cloth simulation where a cloth consisting of 20K triangles drapes around a sphere. Figure (a) shows a snapshot of the simulation. Figure (b) shows the triangles in PSCS.*

We have tested our system for computing self-collisions in a cloth simulator. The cloth is represented as a rectangular grid with nearly 20K triangles. The simulation is performed on the CPU and could be easily implemented on the GPU using pixel shaders. Our pruning algorithm is able to compute self-collisions within 20 - 30 ms, on average.

5.6.3 FAR

We have implemented FAR on a Dell Precision workstation with a 2.8 GHz Xeon processor, 1 GB of main memory, and an NVIDIA GeForce FX 5950 Ultra graphics card. We use a viewport resolution of 1400×1400 for performing all the computations. We improve the rendering throughput by using vertex arrays and use `GL_NV_occlusion_query` for performing the visibility queries.

We have tested our algorithm on three complex scenes and have compared its culling performance and accuracy with some prior approaches.

Dynamically generated breaking objects: The scene consists of a dragon model, initially with 112K polygons, and a bunny model, initially with 35K polygons, as shown in Fig. 5.9. In this simulation, the bunny falls on the dragon, causing the dragon to break into many pieces over the course of the simulation. Each piece is treated as a separate object for collision detection. Eventually hundreds of new objects are introduced into the environment. We perform collision culling to compute which object pairs are in close proximity. It takes about 35 ms towards the beginning of the simulation, and about 50 ms at the end, when the number of objects in the scene is much higher.

Interference computation between complex models: In this scene, we compute all the overlapping-triangle pairs between a 68K-triangle bunny that is moving with respect to another bunny, also with 68K triangles. The bunnies are deeply penetrating, and the intersection boundary consists of 2,000 - 4,000 triangle pairs. In this case, the accuracy of FAR equals that of a CPU-based algorithm using 32-bit IEEE floating-point arithmetic.

Multiple objects with nonrigid motion: This scene consists of a nonrigid simulation in which leaves fall from a tree, as shown in Fig. 5.1. We compute collisions among the leaves of the tree as well as between the leaves and branches of the tree. Each leaf is represented using 156 triangles and the complete environment consists of 40K triangles. The average collision detection time is 35 ms per time step.



Figure 5.9: *Breaking-Object Scene: In this simulation, the bunny model falls on the dragon which eventually breaks into hundreds of pieces. FAR computes collisions among the new pieces of small objects introduced into the environment and takes 30 to 60 msec per frame.*

5.7 Analysis and Limitations

In this section, we analyze the various factors that affect the accuracy and performance of our algorithms. We also discuss the limitations of our algorithms.

5.7.1 Performance Analysis

The performance of our algorithms is based mainly on the underlying pruning algorithm in CULLIDE. Therefore we have tested the performance of our pruning algorithm in CULLIDE on different benchmarks. Its overall performance is governed by some key parameters. These include the following benchmarks:

- **Number of Objects:** Our object-level pruning algorithm exhibits linear-time performance in our benchmarks. We have performed a timing analysis by varying the number of deformable objects, and Fig. 5.10 summarizes the results. In our simulations, we have observed that the pruning algorithm requires only a few iterations in order to converge (typically, it is two). Also, each iteration reduces the size of the PCS. Therefore the visibility-based pruning algorithm traverses a smaller list of objects during subsequent iterations.

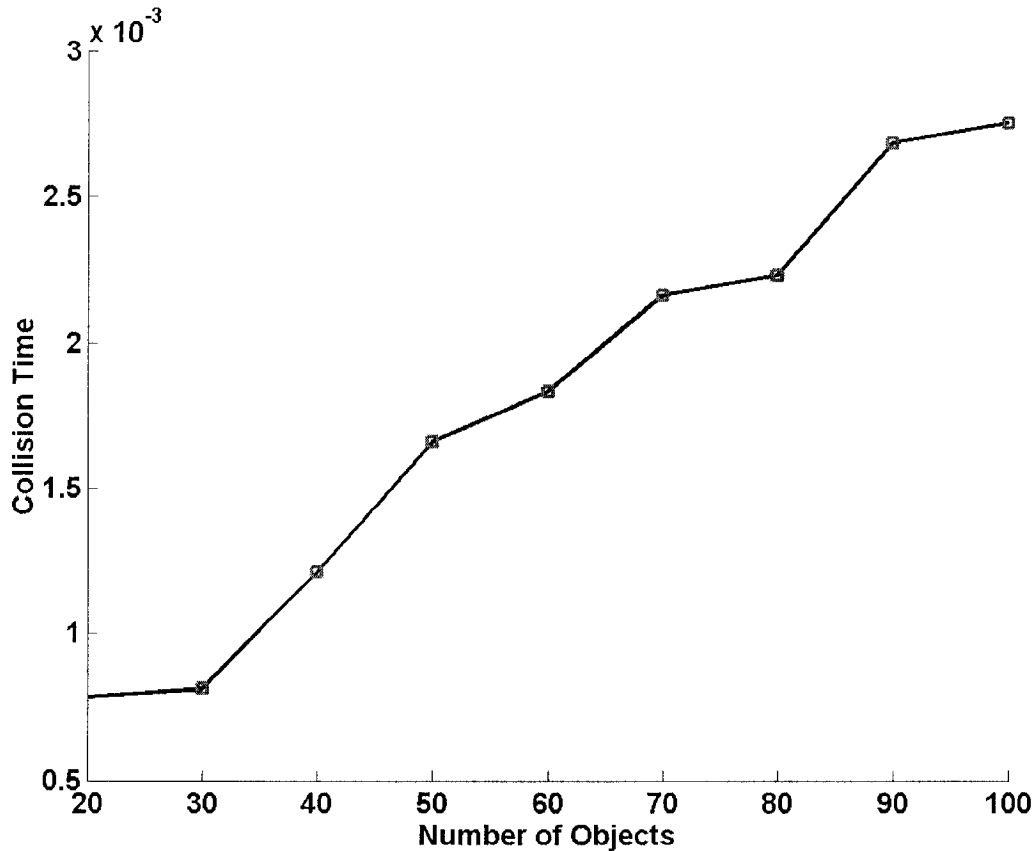


Figure 5.10: *Number of Objects vs. Average Collision-Pruning Time using CULLIDE: This graph highlights the relationship between the number of objects in the scene and the average collision-pruning time (object pruning and subobject/triangle pruning). Each object is composed of 200 triangles. The graph indicates that the collision-pruning time is linear to the number of objects.*

- Triangle Count per Object:** The performance of our pruning algorithm depends upon the triangle count of the potentially intersecting objects. We have tested our system with simulations consisting of 100 deformable objects and varying the triangle count of the objects. Fig. 5.11 summarizes the results. The graph indicates a linear relationship between the polygon count and the average collision-query time. Moreover, the number of polygons per object is much higher than the number of objects in the scene.

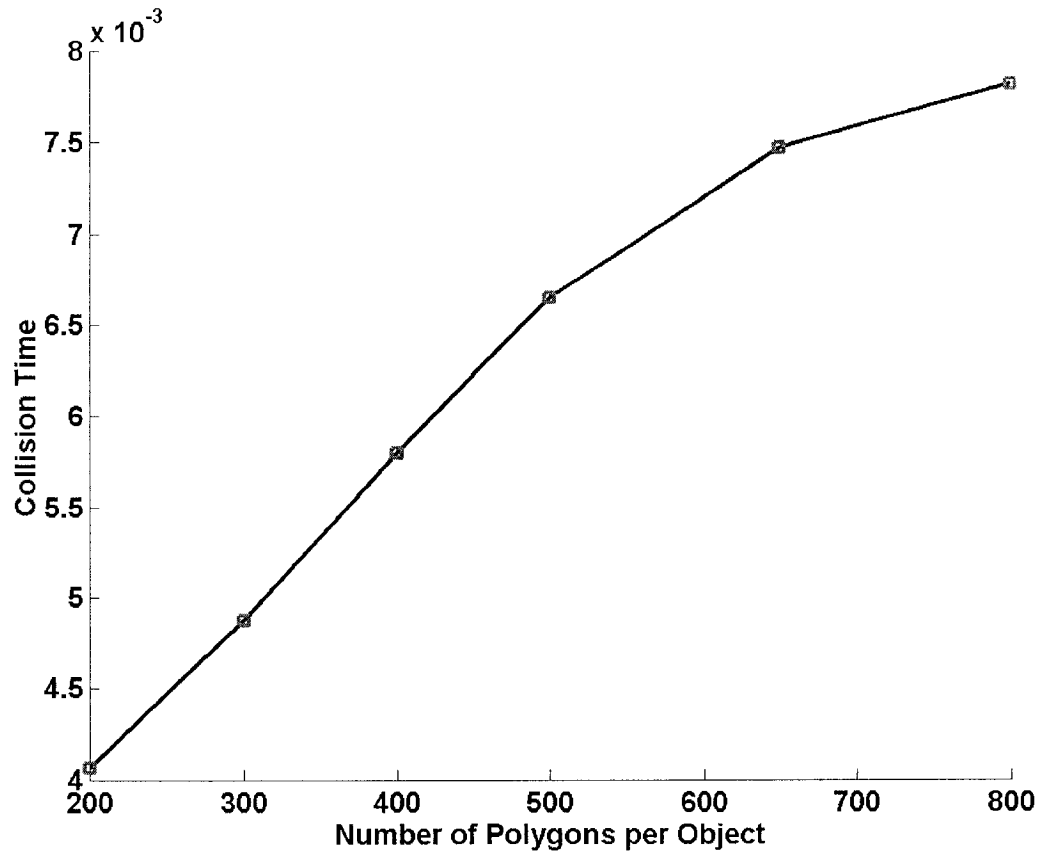


Figure 5.11: *Polygons per Object vs. Average Collision-Query Time using CULLIDE: This graph shows the linear relationship between the number of polygons per object and the average collision-pruning time. This scene is composed of 100 deforming cylinders and has a density of 1 - 2%. The collision-pruning time is averaged over 500 frames and at an image-space resolution of 800×800 .*

- Accuracy and Image-Space Resolution :** The accuracy of the overall algorithm is governed by the image-space resolution. Typically a higher resolution leads to higher fill-rate requirements in terms of rendering the primitives and bounding boxes and performing occlusion queries. A lower image-space resolution can improve the query time but can miss intersections between two objects, whose boundaries are touching tangentially or have a very small penetration. Figure 5.12 highlights the relationship between the collision-pruning time and the image-space resolution.

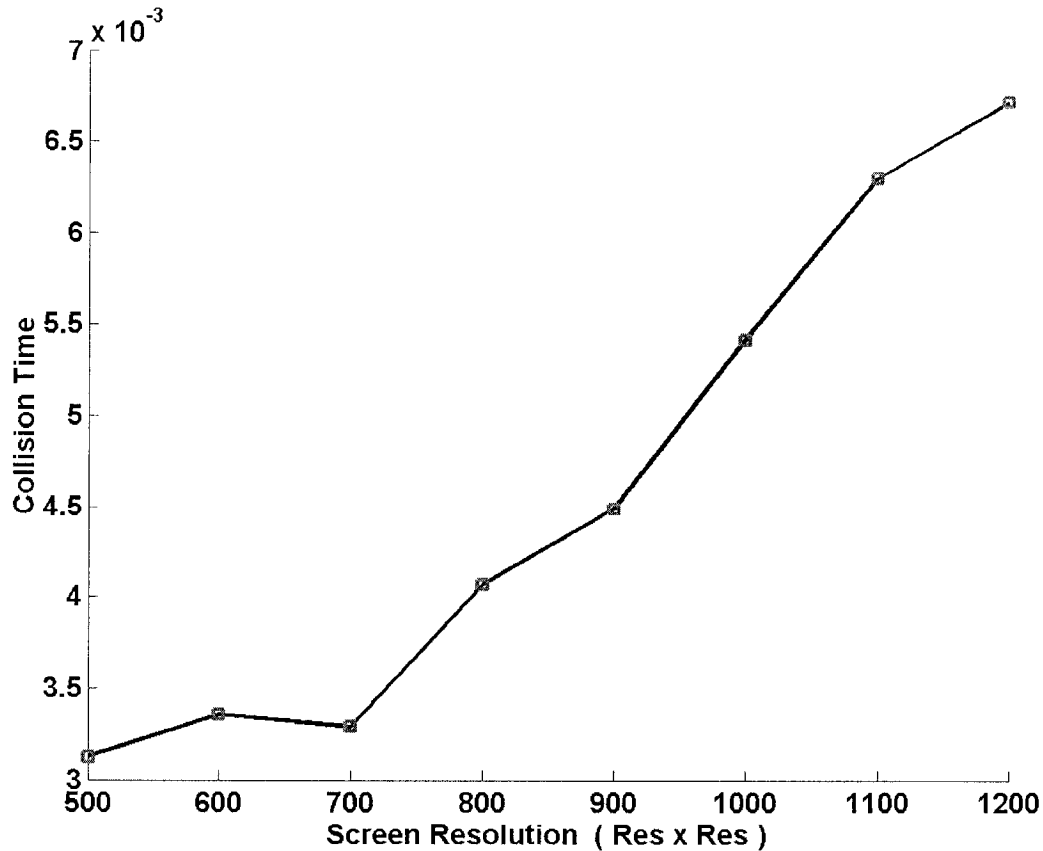


Figure 5.12: *Image-Space Resolution vs. Average Collision-Query Time using CULLIDE:* This graph indicates the linear relationship between the screen resolution and the average collision-query time. The scene consists of 100 deformable cylinders and each object is composed of 200 triangles.

- **Output Size:** The performance of any collision detection algorithm varies as a function of the output size (i.e., the number of overlapping triangle pairs). In our case, the performance varies as a linear function of the size of the PCS after object-level pruning and subobject-level pruning as well as of the number of triangle pairs that need to be tested for exact contact. In case two objects have a deep intersection or penetration, the output size can be large and therefore the size of each PCS can be large as well.
- **Rasterization Optimizations:** The performance of our algorithms is accelerated using the rasterization optimizations discussed in Section 5.5.7. We have

used AABBs with orthographic projections for our pruning algorithms. We have used immediate mode for rendering the models and breakable objects, and used `GL_NV_occlusion_query` to maximize the performance.

5.7.2 Pruning Efficiency

Our approach for collision detection is based on pruning techniques. Its overall performance depends on the input complexity, the relative configuration of different objects in the scene as well as pruning efficiency of the object-level and subobject level algorithms. Most pruning algorithms based on bounding-volume hierarchies can take a long time for parallel close-proximity scenarios (Gottschalk et al., 1996a) (e.g., two concentric spheres with nearly the same radius). Our algorithm performs pruning at the triangle level and works well in these configurations. It should be noted that the pruning efficiency depends largely upon the choice of view direction for orthographic projection. Certain view directions may not provide sufficient pruning, as a larger number of objects may be partially visible from these views. One possible solution for avoiding such configurations is to select the directions for orthographic projection randomly.

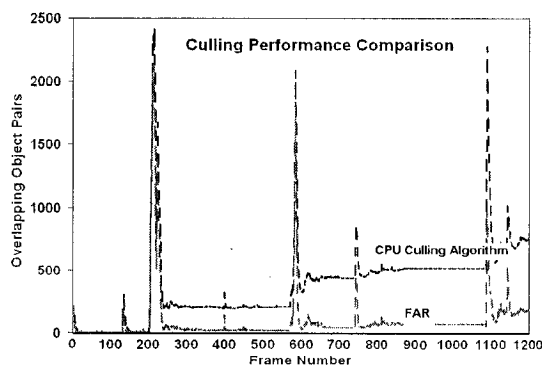


Figure 5.13: *Relative Culling Performance on Breaking-Objects Scene: This graph highlights the improved culling performance of our algorithm as compared to a CPU-based culling algorithm (I-COLLIDE) that uses AABBs (axis-aligned bounding boxes) to cull away non-overlapping pairs. FAR reports 6.9 times fewer pairs over the entire simulation.*

We compared the pruning efficiency of our GPU-based reliable culling algorithm with

an implementation of the sweep-and-prune algorithm available in I-COLLIDE (Cohen et al., 1995). The comparison was performed on the environment with dynamically generated breaking objects. The sweep-and-prune algorithm computes an axis-aligned bounding box (AABB) for each object in the scene and checks all the AABBs for pairwise overlaps. Fig. 5.13 shows the comparison between the culling efficiency of AABB-based algorithm and that of FAR. Overall, FAR returns 6.9 times fewer overlapping pairs. This reduction occurs mainly because FAR uses much tighter bounding volumes (i.e., the union of OBBs) for an object as compared to an AABB and is able to cull away more primitive pairs.

As CULLIDE and S-CULLIDE miss collisions, and the amount of inaccuracy in their computations is not bounded, it is not possible to compare their pruning efficiencies with those of other algorithms.

5.7.3 Precision

The precision of CULLIDE and S-CULLIDE is governed mainly by the image resolution and frame-buffer precision. However, our reliable culling algorithm FAR is conservative and its precision is governed by that of the reliable VO queries. The accuracy of the reliable culling algorithm is equivalent to that of the floating-point hardware (e.g., 32-bit IEEE floating-point) inside the GPUs used for performing transformations and rasterization. The precision of FAR is not governed by viewport resolution or frame-buffer precision.

We have compared our algorithm with an optimized GPU implementation of CULLIDE on the environment illustrating interference computation between complex models. Our implementation runs nearly 3 times slower due to the overhead of rasterizing bounding boxes instead of triangles. However, as shown in Fig. 5.14, CULLIDE (like other GPU-based collision detection algorithms) misses several interferences and may lead to inaccurate simulations.

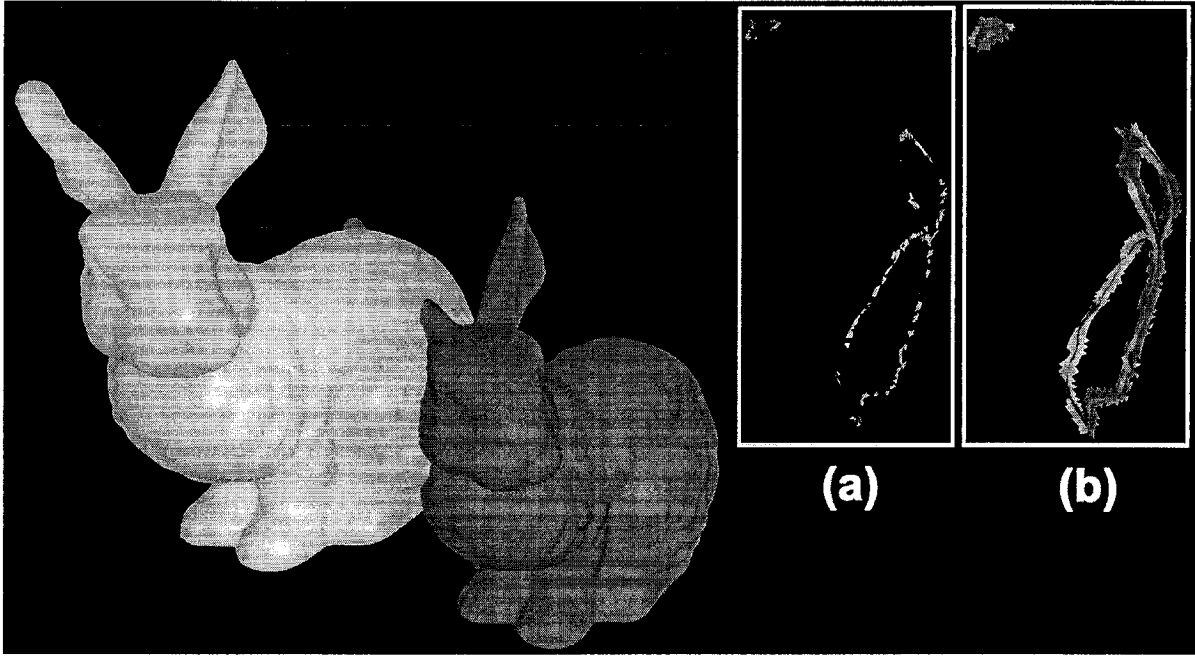


Figure 5.14: *Reliable interference computation: This image highlights the intersection set between two bunnies, each with 68K triangles. The top right image (b) shows the output of FAR and the top left image (a) highlights the output of CULLIDE running at a resolution of 1400×1400 . CULLIDE misses many collisions due to the viewport resolution and sampling errors.*

5.7.4 Comparison with Other Approaches

Collision detection is well studied in the literature, and a number of algorithms and public-domain systems are available. However, none of the earlier algorithms provide the same capabilities or features as our algorithm based on PCS computation. As a result, we have not performed extensive timing comparisons with the earlier systems. We just compare some of the features of our approach with those of the earlier algorithms.

Object-Space Algorithms: Algorithms based on sweep-and-prune are known for N-body collision detection (Cohen et al., 1995). They have been used in I-COLLIDE, V-COLLIDE, SWIFT, SOLID, and other systems. However, these algorithms were designed for rigid bodies and compute a tight-fitting AABB for each object using incremental methods, followed by sorting their projections of AABBs along each axis. It is not clear whether they can perform real-time collision detection on large environments composed of deformable models. On the other hand, our algorithm performs two passes

on the object list to compute the PCS. We expect that our PCS-based algorithm to be more conservative as compared to sweep-and-prune. Furthermore, the accuracy of our approach is governed by the image-space resolution.

A number of hierarchical algorithms have been proposed to check two highly tessellated models for overlap, and some optimized systems (e.g., RAPID, SOLID, and QuickCD) are also available. They involve considerable preprocessing and memory overhead in generating the hierarchy and will not work well on deformable models or objects with changing topologies. We have compared FAR with a CPU-based implementation SOLID (SOLID, 2002) on the environment with dynamically generated breaking objects.³ SOLID is a publicly available library that uses pre-computed AABB trees for collision culling. As the objects in our scene are dynamically generated and the topologies of existing objects (e.g., dragon) change, we need to compute the hierarchies dynamically. Moreover, as the hierarchies are recomputed, there is an additional overhead of allocating and deallocating memory in SOLID. Ignoring the overhead due to memory, we observed that the pre-computation of data structures for SOLID require 100 - 176 ms per frame. These timings do not include the pruning time. On the contrary, we are able to compute all collisions up to floating-point precision within 50ms (including UOBB-computation time and pruning time).

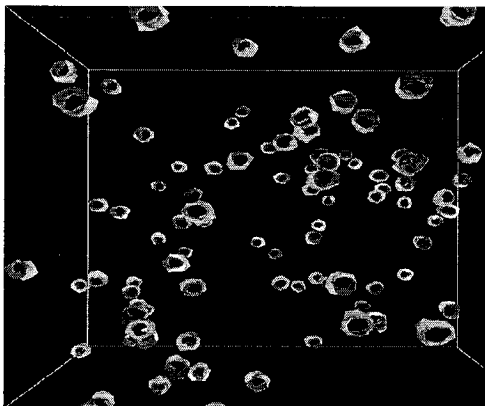
Image-Space Algorithms : These include algorithms based on stencil-buffer techniques as well as distance field computations. Some systems, such as PIVOT, are able to perform other proximity queries, including distance and local penetration computation, whereas our PCS-based algorithm is limited to only checking for interference. However, our algorithm only needs to readback the output of a visibility query and not the entire depth buffer or stencil buffer. This feature significantly improves its performance, especially when we use higher image-space precision. Unlike earlier algorithms, our PCS-based algorithm is applicable to all triangulated 3-D models (and not just closed

³For the sake of fairness, we have not compared CULLIDE with SOLID, as CULLIDE misses interferences whereas SOLID does not.

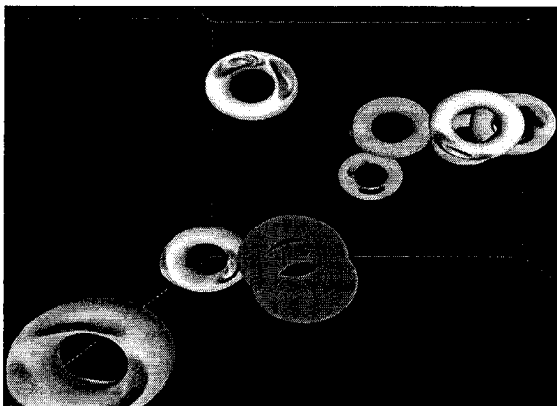
objects) and can handle an arbitrary number of objects in the environment. Also, unlike prior GPU-based collision detection algorithms, our reliable culling algorithm does not miss collisions due to image-sampling or frame-buffer precision errors.

5.7.5 Limitations

Our current approach has some limitations. First, it only checks for overlapping objects and does not provide distance or penetration information. Our reliable collision culling algorithm cannot extend directly to performing reliable self-collisions. In practice, however, it is possible to apply the reliable algorithm for performing reliable self-collisions efficiently in specialized environments such as cloth models.



(a) *Environment 1: This scene consists of 100 dynamically deforming open cylinders moving randomly in a room. Each cylinder is composed of 200 triangles.*



(b) *Environment 2: This scene consists of 10 dynamically deforming torii moving randomly in a room. Each torus is composed of 20000 triangles*



(c) *Environment 3: This scene shows a wired frame of a dragon and a bunny rendered in the following colors: cyan and blue, which highlight triangles in the PCS, and red and white, which illustrate portions not in the PCS. The PCS is computed using CULLIDE. The dragon consists of 250K triangles and the bunny consists of 35K triangles*



(d) *Environment 3: Zoomed view highlighting the exact intersections between the triangles in the PCS computed using CULLIDE. The configuration of the two objects is the same as in Fig. 5.15(c). The cyan and the blue regions are the overlapping triangles of the dragon and the bunny respectively.*

Figure 5.15: *Snapshots of Simulations on Three Complex Environments: CULLIDE takes 4, 8, and 40ms respectively on each benchmark in order to perform collision queries on a GeForce FX 5800 Ultra with an image resolution of 800×800 .*

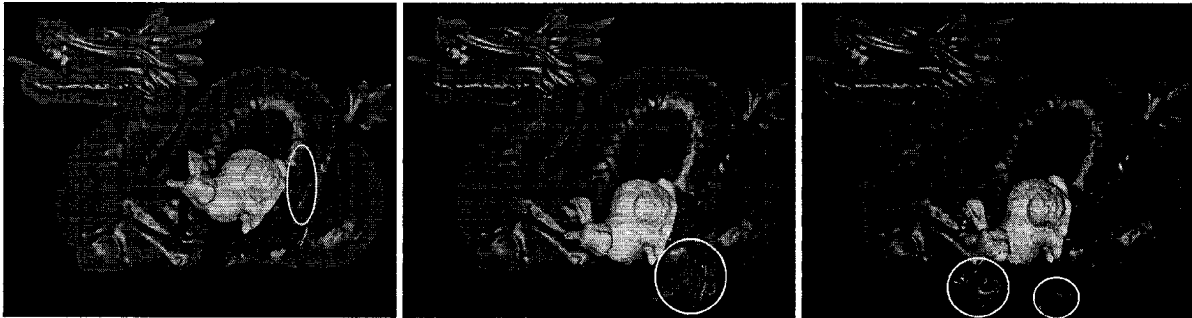


Figure 5.16: *Environment with Breakable Objects: As the bunny (with 35K triangles) falls through the dragon (with 250K), the number of objects in the scene (shown with a yellow outline) and the triangle count within each object change. CULLIDE computes all the overlapping triangles during each frame. The average collision-query time is 35 ms per frame.*

Chapter 6

Conclusions

In this thesis, we have designed efficient visibility-based algorithms for solving three different problems: walkthroughs, shadow generation, and collision detection. Our algorithms utilize the occlusion queries available on commodity GPUs for performing visibility computations. We have applied these algorithms to complex environments composed of hundreds of thousands of primitives. We have also compared the performance of some of our algorithms to prior state-of-the-art algorithms. In some cases, we obtained significant increases in speed as compared to earlier algorithms. Moreover, our algorithms have many advantages:

1. **Generality:** They make no assumptions about the scenes and are applicable to all complex environments.
2. **Accuracy:** Our algorithms perform conservative visibility computations up to
 - screen-space image precision for walkthrough and shadow computations, and
 - 32-bit IEEE floating-point precision for collision computation.
3. **Low Bandwidth:** Our algorithms involve no depth-buffer readback from the graphics card. The bandwidth requirements of our algorithms vary as a function based on the number of primitives in the scenes and, in practice, is very low (e.g., a few kilobytes per frame).

4. **Significant Occlusion Culling:** As compared to earlier approaches, our algorithms cull away a higher percentage of primitives
 - not visible from the current viewpoint in the cases of walkthrough and shadow computations, and
 - not colliding from the current viewpoint in the case of collision computations.
5. **Practicality:** Our algorithms can be implemented on commodity graphics processors, and only assume hardware support for the occlusion query, which has become widely available. Furthermore, we obtain significant improvements in frame rate as compared to earlier algorithms.

Our algorithms do, however, involve some limitations:

1. **Environments:** Our visibility-based algorithms work well for complex 3-D environments exhibiting
 - high depth complexity for walkthrough and shadow computation, and
 - deforming and dynamically changing topologies for collision computation.
2. **Latency:** Our parallel algorithms for walkthrough and shadow generation involve a frame of latency in addition to double buffering. These algorithms are well suited for latency-tolerant applications.
3. **Precision:** The accuracy of our shadow culling and self-collision pruning algorithms are limited by image precision. Their accuracy can be improved by using a reliable algorithm similar to our reliable collision detection algorithm.
4. **Pair Computation:** Our algorithms compute potential sets of primitives:
 - potentially visible sets of primitives for walkthrough computation,
 - potential shadow casters and shadow receivers for shadow computation, and

- potentially colliding sets of primitives for collision computation.

Some algorithms require the computation of pairs of associated primitives. For example, a shadow-polygon-clipping algorithm requires the shadow receivers associated with each shadow caster to generate shadow polygons, and a collision detection algorithm requires the pairs of potentially colliding primitives. As our algorithms compute sets of primitives, they need to be integrated with other algorithms for generating pairs of associated primitives.

5. **Artifacts:** Our interactive algorithms for walkthrough and shadow generation have been integrated with static levels-of-detail. Using static LODs may lead to popping artifacts at high LOD-error threshold due to switching between approximations.
6. **Coherence:** Our real-time culling algorithms for walkthrough and shadow generation assume high spatial coherence between successive frames. The culling efficiency may not be high if the camera position changes significantly.

In the following sections, we summarize our algorithms and describe avenues for future investigation.

6.1 Walkthroughs

We have presented a new occlusion culling algorithm based on occlusion switch and used it for rendering massive models at interactive rates. Each occlusion switch is composed of two GPUs and uses the hardware occlusion query that is becoming widely available on commodity GPUs for computing the potentially visible set of geometric primitives. We have combined the occlusion culling algorithm with static levels-of-detail (LODs) and used it for interactive walkthrough of complex environments. The static LODs are computed as a preprocess. We present a novel clustering algorithm for

generating a unified scene-graph hierarchy used for generating LODs and performing occlusion culling. We highlighted the performance of our occlusion culling algorithm on three complex environments. We have observed 2 - 3 times improvement in frame rate over earlier approaches. The culling performance of the algorithm is further improved by using a subobject hierarchy and the improved algorithm is used for interactive shadow generation (Govindaraju et al., 2003a).

6.1.1 Future Work

Many research opportunities lie ahead. A low-latency network implementation is highly desirable to maximize the performance achieved by our parallel occlusion culling algorithm. One possibility is to use raw GM sockets over Myrinet. We are also exploring the use of a reliable protocol over UDP/IP. Our current implementation loads the entire scene graph and object LODs on each PC. Due to limitations on the main memory, we would like to use out-of-core techniques that use a limited memory footprint. One possibility for a simple out-of-core solution can be obtained by using additional frames of latency, during which visible geometric primitives can be pre-fetched from the disk. Moreover, the use of static LODs and HLODs can lead to popping artifacts as the rendering algorithm switches between different approximations. One possibility is to use view-dependent simplification techniques in order to alleviate these artifacts (Yoon et al., 2003). Finally, we would like to apply our algorithm to other complex environments.

6.2 Shadow Generation

We have presented a shadow culling algorithm for interactive shadow generation in complex environments with a moving light source. Our algorithm can be used for generating shadows with sharp edges and for reducing the aliasing artifacts present in pure image-precision approaches. Our algorithm uses level-of-detail techniques for

generating shadows at interactive rates. We have presented an LOD-selection metric that attempts to reduce artifacts in shadows due to switching of LODs. We have integrated our algorithm with a hybrid shadow generation technique and applied it to three large models. Our preliminary results are very encouraging.

We have also presented an improved algorithm for PVS computation, which can be useful for other rendering applications. Our cross-culling algorithm can be used for accelerating the performance of a pure-shadow-volume-based approach such as CC shadow volumes (Lloyd et al., 2004).

6.2.1 Future Work

There are many promising directions for further investigation. Our current approach handles only point and directional light sources. We would also like to explore ways to handle omni-directional or multiple light sources by using additional graphics processors. We want to develop an out-of-core system that does not load the entire scene graph into main memory. Using view-dependent simplification in place of static LODs could reduce popping artifacts. Finally, we would like to extend our algorithms for improving the performance of soft-shadow-generation algorithms.

6.3 Collision Detection

We have presented a novel algorithm for collision detection among multiple deformable objects in a large environment using graphics hardware. Our algorithm is applicable to all triangulated models, makes no assumptions about object motion, and can compute all contacts up to image-space resolution. It uses a novel linear-time PCS-computation algorithm applied iteratively to objects and subobjects. The PCS is computed using image-space visibility queries widely available on commodity graphics hardware. It needs to readback the results of a query, not the frame-buffer or depth

buffer.

Our algorithm can also compute self-collisions in general deformable models and does not require mesh-connectivity information. Further, our algorithm can perform reliable collision detection among multiple objects with little computational overhead. Our reliable algorithm overcomes a major limitation of earlier GPU-based collision detection algorithms and is able to perform reliable interference queries. Furthermore, the culling efficiency of our algorithm is higher as compared to prior CPU-based algorithms that use AABBs or spheres for collision culling. We have demonstrated its performance in complex scenarios where objects undergo rigid and nonrigid motion.

Our algorithm can be applied to other problems such as continuous collision detection (Redon et al., 2003) and shadow volume clamping (Lloyd et al., 2004).

6.3.1 Future Work

There are many avenues for future work. Our PCS computation algorithm only computes sets of potentially colliding objects. We would like to extend our algorithm to compute pairs of colliding objects. We would like both to use our PCS-based collision detection for more applications and to evaluate its impact on the accuracy of the overall simulation. Our current self-collision algorithm computes collisions among geometric primitives up to image-precision. A reliable self-collision algorithm for handling general deformable models would be very useful. Further, we are exploring reliable self-collision detection algorithms for handling specialized environments such as cloth models, etc. We are also investigating the use of the new programmability features of graphics hardware for improving our algorithms as described below.

Desirable Hardware Features: We propose a simple architecture for the graphics pipeline for accelerating the performance of our algorithm. The modified architecture requires the following characteristics:

- **Precision:** In order to obtain floating-point precision, the graphics pipeline should support floating-point depth buffers. However, it is important to note that the viewport resolution is mainly responsible for the sampling errors than the depth-buffer precision. Therefore floating-point depth buffers alone cannot solve the sampling problem.
- **Rasterization Rules:** We set a state in which the following rasterization rules are used in order to overcome the viewport-resolution problems.
 - A fragment is generated for each pixel that a triangle touches.
 - For each pixel, depth is computed at all four corner samples of the rectangular pixel. A *depth set* function is applied to the four depth samples, and one of the four values is output as the depth of the current fragment. The depth set function could be one of $\{max, min\}$ and is specified as a state before rasterizing a primitive. The function *max* computes the maximum value of the four depth samples, and *min* computes the minimum value of the four depth samples.

The above rules are sufficient for designing an algorithm ensuring floating-point precision for interference computations. In the pseudocode described in Section 5.5.7, while rendering a primitive to the frame buffer, we set the depth set function to *min* and the depth function to *GL_LEQUAL*. This operation ensures that for each pixel touched by a primitive, we compute the minimum depth of *all* points of the primitive that project onto the pixel. While testing the fully visible status of a primitive, the depth set function is set to *max*. This operation ensures that we test whether the maximum depth of all points of a primitive that project onto the pixel is fully visible. It is easy to see that these two operations can be used for testing conservatively whether one primitive interferes with another.

Most graphics hardware implementations involve tile-based rasterization (Kelleher,

1998; McCormack and McNamara, 2000; Morein, 2000; Akenine-Moller and Strom, 2003). All the pixels covered by a primitive and within a tiled region (e.g., a region consisting of 4×4 pixels) are computed before moving to the next tile. As adjacent pixels share common sample points, it is possible to design a simple architecture computing the depth at a sample point (e.g., the left corner of a pixel) and depth values at the samples covering the top and right corners of a tile. A simple hardware can be used for computing the max or min values of these fragments in a tile and returning the sample depths.

A simple hardware implementation requires the computation of few more samples than the actual number of samples in a normal rasterization pipeline. However, the overhead of this computation can be minimized by using additional hardware such as pixel caches.

BIBLIOGRAPHY

- Adelson, E. and Bergen, J. (1991). The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, pages 3–20. MIT Press.
- Aila, T. and Akenine-Möller, T. (2004). A hierarchical shadow volume algorithm. In *Proceedings of Graphics Hardware 2004*.
- Airey, J., Rohlf, J., and Brooks, F. (1990). Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50.
- Akenine-Möller, T. and Strom, J. (2003). Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808.
- Aliaga, D., Cohen, J., Wilson, A., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Baker, E., Bastos, R., Whitton, M., Brooks, F., and Manocha, D. (1999). MMR: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 199–206.
- Aliaga, D. and Lastra, A. (1999). Automatic image placement to provide a guaranteed frame rate. In *Proc. of ACM SIGGRAPH*.
- Atherton, P., Weiler, K., and Greenberg, D. (1978). Polygon shadow generation. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 275–281.
- Baciu, G. and Wong, S. (2002). Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*.
- Baciu, G., Wong, S., and Sun, H. (1998). Recode: An image-based collision detection algorithm. *Proc. of Pacific Graphics*, pages 497–512.
- Barequet, G., Chazelle, B., Guibas, L., Mitchell, J., and Tal, A. (1996). Boustrophedon: A hierarchical representation of surfaces in 3D. In *Proc. of Eurographics'96*.
- Bartz, D., Meibner, M., and Huttner, T. (1999). OpenGL assisted occlusion culling for large polygonal models. *Computer and Graphics*, 23(3):667–679.
- Baxter, B., Sud, A., Govindaraju, N., and Manocha, D. (2002). GigaWalk: Interactive walkthrough of complex 3D environments. *Proc. of Eurographics Workshop on Rendering*, pages 203–214.
- Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990). The r*-tree: An efficient and robust access method for points and rectangles. *Proc. SIGMOD Conf. on Management of Data*, pages 322–331.

- Bergeron, P. (1985). Shadow volumes for non-planar polygons. In *Graphics Interface '85 Proceedings*, pages 417–418.
- Bittner, J., Havran, V., and Slavík, P. (1998). Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98 (CGI'98)*, pages 207–219. IEEE.
- Bittner, J. and Wonka, P. (2003). Visibility in computer graphics. In *Journal of Environment and Planning*.
- Blinn, J. (1988). Jim blinn's corner: Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86.
- Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. (2003). Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 22(3).
- Brabec, S., Annen, T., and Seidel, H. (2001). Hardware-accelerated rendering of antialiased shadows. In *Proc. of Computer Graphics International*.
- Brabec, S., Annen, T., and Seidel, H. (2002). Shadow mapping for hemispherical and omnidirectional light sources. In *Proc. of Computer Graphics International*.
- Brabec, S. and Seidel, H. (2003). Shadow volumes on programmable graphics hardware. *Proc. of Eurographics*.
- Brotman, L. S. and Badler, N. I. (1984). Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):71–81.
- Buehler, C., Bosse, M., McMillan, L., Gortler, S., and Cohen, M. (2001). Unstructured lumigraph rendering. In *Proceedings of ACM Siggraph*.
- Chan, E. and Durand, F. (2004). An efficient hybrid shadow rendering algorithm. In *Proceedings of the Eurographics Symposium on Rendering*. Eurographics Association.
- Chin, N. and Feiner, S. (1989). Near real-time shadow generation using BSP trees. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 99–106.
- Chrysanthou, Y. and Slater, M. (1995). Shadow volume BSP trees for computation of shadows in dynamic scenes. In *1995 Symposium on Interactive 3D Graphics*, pages 45–50.
- Clark, J. (1976). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554.
- Cohen, J., Lin, M., Manocha, D., and Ponamgi, M. (1995). I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196.

- Cohen, J., Manocha, D., and Olano, M. (1997). Simplifying polygonal models using successive mappings. In *Proc. of IEEE Visualization*, pages 395–402.
- Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, Jr., F. P., and Wright, W. (1996). Simplification envelopes. In Rushmeier, H., editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 119–128. ACM SIGGRAPH, Addison Wesley. held in New Orleans, Louisiana, 04-09 August 1996.
- Cohen, J. D. and Manocha, D. (2003). Model simplification. In *Handbook of Visualization*.
- Cohen-Or, D., Chrysanthou, Y., Durand, F., Greene, N., Koltun, V., and Silva, C. (2001a). Visibility, problems, techniques and applications. *SIGGRAPH Course Notes # 30*.
- Cohen-Or, D., Chrysanthou, Y., and Silva, C. (2001b). A survey of visibility for walk-through applications. *SIGGRAPH Course Notes # 30*.
- Coorg, S. and Teller, S. (1996). A spatially and temporally coherent object space visibility algorithm. Technical Report TM-546, Laboratory of Computer Science, MIT.
- Coorg, S. and Teller, S. (1997). Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*.
- Crow, F. (1977). Shadow algorithms for computer graphics. volume 11, pages 242–248.
- D03 depthbounds (2003). Ext_depth_bounds_test specification. http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_depth_bounds_test.txt.
- Darsa, L., Costa, B., and Varshney, A. (1998). Walkthroughs of complex environments using image-based simplification. *Computer and Graphics*, 22(1):55–69.
- Decoret, X., Schaufler, G., Sillion, F., and Dorsey, J. (1999). Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3).
- Durand, F., Drettakis, G., Thollot, J., and Puech, C. (2000). Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, pages 239–248.
- El-Sana, J., Sokolovsky, N., and Silva, C. (2001). Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*.
- Erikson, C. and Manocha, D. (1998). Simplification culling of static and dynamic scene graphs. Technical Report TR98-009, Department of Computer Science, University of North Carolina.
- Erikson, C. and Manocha, D. (1999). GAPS: General and automatic polygon simplification. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

- Everitt, C. and Kilgard, M. (2002). Practical and robust stenciled shadow volumes for hardware-accelerated rendering. In *SIGGRAPH 2002 Course Notes*, volume 31.
- Felzenszwalb, P. and Huttenlocher, D. (1998). Efficiently computing a good segmentation. In *Proceedings of IEEE CVPR*, pages 98–104.
- Fernando, R., Fernandez, S., Bala, K., and Greenberg, D. (2001). Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH 2001*, pages 387–390.
- Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1990). *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA. second edition.
- Garland, M. and Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *Proc. of ACM SIGGRAPH*, pages 209–216.
- Gottschalk, S., Lin, M., and Manocha, D. (1996a). OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180.
- Gottschalk, S., Lin, M. C., and Manocha, D. (1996b). OBB-tree: A hierarchical structure for rapid interference detection. *Computer Graphics*, pages 171–180.
- Govindaraju, N., Lloyd, B., Wang, W., Lin, M., and Manocha, D. (2004). Fast computation of database operations using graphics processors. *Proc. of ACM SIGMOD*.
- Govindaraju, N., Lloyd, B., Yoon, S., Sud, A., and Manocha, D. (2003a). Interactive shadow generation in complex environments. *Proc. of ACM SIGGRAPH/ACM Trans. on Graphics*, 22(3):501–510.
- Govindaraju, N., Redon, S., Lin, M., and Manocha, D. (2003b). CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32.
- Govindaraju, N., Sud, A., Yoon, S., and Manocha, D. (2002). Interactive visibility culling in complex environments with occlusion-switches. Technical Report CS-02-027, University of North Carolina. Appeared in *Proc. of ACM Symposium on Interactive 3D Graphics*.
- Govindaraju, N., Sud, A., Yoon, S., and Manocha, D. (2003c). Interactive visibility culling in complex environments with occlusion-switches. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 103–112.
- Greene, N. (2001). Occlusion culling with optimized hierarchical z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY*, # 30.
- Greene, N., Kass, M., and Miller, G. (1993). Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238.

- Guha, S., Krishnan, S., Munagala, K., and Venkatasubramanian, S. (2003). Application of the two-sided depth test to csg rendering. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 177–180. ACM Press.
- Hasenfratz, J.-M., Lapierre, M., Holzschuch, N., and Sillion, F. (2003). A survey of real-time soft shadows algorithms. In *Eurographics*. Eurographics, Eurographics. State-of-the-Art Report.
- Heidelberger, B., Teschner, M., and Gross, M. (2003). Real-time volumetric intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*.
- Heidmann, T. (1991). Real shadows real time. *IRIS Universal*, (18).
- Heidrich, W. and Seidel, H. P. (1999). Realistic hardware-accelerated shading and lighting. In *Proc. of ACM SIGGRAPH*, pages 171–178.
- Held, M., Klosowski, J., and Mitchell, J. S. B. (1996). Real-time collision detection for motion simulation within complex environments. In *Proc. ACM SIGGRAPH'96 Visual Proceedings*, page 151.
- Hillesland, K., Salomon, B., Lastra, A., and Manocha, D. (2002). Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, University of North Carolina.
- Hoff, K., Zaferakis, A., Lin, M., and Manocha, D. (2001). Fast and simple 2d geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 145–148.
- Hoppe, H. (1996). Progressive meshes. In *Proc. of ACM SIGGRAPH*, pages 99–108.
- Hoppe, H. (1997). View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, pages 189–198.
- Hoppe, H. (1998). Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization Conference Proceedings*, pages 35–42.
- Hubbard, P. M. (1993). Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*.
- Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., and Zhang, H. (1997a). Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, pages 1–10.
- Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., and Zhang, H. (1997b). Accelerated occlusion culling using shadow frusta. In *ACM Symposium on Computational Geometry*, pages 1–10.
- Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. (2001). Wiregl: A scalable graphics system for clusters. *Proc. of ACM SIGGRAPH*.

- Jacobson, V. (1988). Congestion avoidance and control. *Proc. of ACM SIGCOMM*, pages 314–329.
- Jeschke, S. and Wimmer, M. (2002). Textured depth mesh for real-time rendering of arbitrary scenes. In *Proc. Eurographics Workshop on Rendering*.
- Jimenez, P., Thomas, F., and Torras, C. (2001). 3d collision detection: A survey. *Computers and Graphics*, 25(2):269–285.
- Kelleher, B. (1998). Pixelvision architecture. Technical Report 1998-013, Digital Systems Research Center.
- Kim, Y., Lin, M., and Manocha, D. (2002a). DEEP: an incremental algorithm for penetration depth computation between convex polytopes. *Proc. of IEEE Conference on Robotics and Automation*, pages 921–926.
- Kim, Y., Otaduy, M., Lin, M., and Manocha, D. (2002b). Fast penetration depth computation for physically-based animation. *Proc. of ACM Symposium on Computer Animation*.
- Klosowski, J., Held, M., Mitchell, J., Sowizral, H., and Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37.
- Klosowski, J. and Silva, C. (2001). Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):365–379.
- Knott, D. and Pai, D. (2003). Cinder: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, pages 73–80.
- Kruger, J. and Westermann, R. (2003). Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 22(3).
- Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of American Mathematical Society*, 7:48–50.
- Kumar, S., Manocha, D., Garrett, W., and Lin, M. (1999). Hierarchical back-face computation. *Comput. & Graphics*, 23(5):681–692.
- Larsson, T. and Akenine-Moller, T. (2001). Collision detection for continuously deforming bodies. In *Eurographics*.
- Lengyel, E. (2002). The mechanics of robust stencil shadows. *Gamasutra*. http://www.gamasutra.com/features/20021011/lengyel_01.htm.
- Levoy, M. and Hanrahan, P. (1996). Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42.

- Lin, M. and Gottschalk, S. (1998). Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*.
- Lin, M. and Manocha, D. (2003). Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.
- Lloyd, B., Wendt, J., Govindaraju, N., and Manocha, D. (2004). Cc shadow volumes. Technical report, University of North Carolina, Department of Computer Science.
- Lombardo, J. C., Cani, M.-P., and Neyret, F. (1999). Real-time collision detection for virtual surgery. *Proc. of Computer Animation*.
- Luebke, D. and Erikson, C. (1997). View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*.
- Luebke, D. and Georges, C. (1995). Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA.
- Max, N. and Ohsaki, K. (1995). Rendering trees from precomputed Z-buffer views. In *Eurographics Rendering Workshop 1995*.
- McCool, M. (2000). Shadow volume reconstruction from depth maps. *ACM Trans. on Graphics*, 19(1):1–26.
- McCormack, J. and McNamara, R. (2000). Tiled polygon traversal using half-plane edge functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 15–21. ACM Press.
- McGuire, M., Hughes, J., Egan, K., Kilgard, M., and Everitt, C. (2003). Fast, practical and robust shadows. Technical report, NVIDIA. http://developer.nvidia.com/object/fast_shadow_volumes.html.
- McMillan, L. and Bishop, G. (1995). Plenoptic modeling: An image-based rendering system. In *Proc. of ACM SIGGRAPH*, pages 39–46.
- Meissner, M., Bartz, D., Huttner, T., Muller, G., and Einighammer, J. (2002). Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. *Computer and Graphics*.
- Moller, T. (1997). A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2).
- Morein, S. (2000). ATI Radeon HyperZ technology. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Hot3D Proceedings*.
- Myszkowski, K., Okunev, O. G., and Kunii, T. L. (1995). Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512.

- NVocclusion Query (2001). Nv_occlusion_query specification.
[http://www.nvidia.com/dev_content/nvopenglspecs/
 GLNV_occlusion_query.txt](http://www.nvidia.com/dev_content/nvopenglspecs/GLNV_occlusion_query.txt).
- Parker, S., Martic, W., Sloan, P., Shirley, P., Smits, B., and Hansen, C. (1999). Interactive ray tracing. *Symposium on Interactive 3D Graphics*.
- Pfister, H., Zwicker, M., van Baar, J., and Gross, M. (2000). Surfels: Surface elements as rendering primitives. *Proc. of ACM SIGGRAPH*.
- Ponamgi, M., Manocha, D., and Lin, M. (1997). Incremental algorithms for collision detection between solid models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):51–67.
- Private Communication with Lloyd (2003). Private communication with Brandon Lloyd.
- Private Communication with Sud (2003). Private communication with Avneesh Sud.
- Purcell, T., Buck, I., Mark, W., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics (Proc. of SIGGRAPH'02)*, 21(3):703–712.
- Quinlan, S. (1994). Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329.
- Raskar, R. and Cohen, M. (1999). Image precision silhouette edges. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 135–140. ACM Press.
- Redon, S., Kim, Y., Lin, M., and Manocha, D. (2003). Fast continuous collision detection for articulated models. Technical Report TR03-038, University of North Carolina, Department of Computer Science.
- Reeves, W., Salesin, D., and Cook, R. (1987). Rendering antialiased shadows with depth maps. In *Computer Graphics (ACM SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291.
- Rossignac, J. and Borrel, P. (1993). Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag.
- Rossignac, J., Megahed, A., and Schneider, B. (1992). Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60.
- Rusinkiewicz, S. and Levoy, M. (2000). Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*.
- Samanta, R., Funkhouser, T., and Li, K. (2001). Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*.

- Samanta, R., Funkhouser, T., Li, K., and Singh, J. P. (2000). Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. *Eurographics/SIGGRAPH workshop on Graphics Hardware*, pages 99–108.
- Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. (2000). Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, pages 229–238.
- Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., and Haeberli, P. (1992). Fast shadows and lighting effects using texture mapping. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252.
- Sen, P., Cammarano, M., and Hanrahan, P. (2003). Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):521–526.
- Shade, J., Gortler, S., wei He, L., and Szeliski, R. (1998). Layered depth images. *Proc. of ACM SIGGRAPH*, pages 231–242.
- Shinya, M. and Fergie, M. C. (1991). Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):131–134.
- Sillion, F., Drettakis, G., and Bodelet, B. (1997). Efficient impostor manipulation for real-time visualization of urban scenery. In *Computer Graphics Forum*, volume 16.
- SOLID (2002). Freesolid: Software library for interference detection. <http://www.win.tue.nl/gino/solid/>.
- Stamminger, M. and Drettakis, G. (2002). Perspective shadow maps. In *Proceedings of ACM SIGGRAPH 2002*, pages 557–562.
- Tadamura, K., Qin, X., Jiao, G., and Nakamae, E. (2001). Rendering optimal solar shadows with plural sunlight depth buffers. *The Visual Computer*, 17(2).
- Teller, S. J. (1992). *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley.
- Udeshi, T. and Hansen, C. (1999). Towards interactive photorealistic rendering of indoor scenes: A hybrid approach. In Lischinski, D. and Larson, G. W., editors, *Rendering Techniques '99*, pages 63–76.
- Vassilev, T., Spanlang, B., and Chrysanthou, Y. (2001). Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics'01)*, 20(3):260–267.
- Wald, I., Slusallek, P., and Benthin, C. (2001). Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, pages 274–285.
- Williams, L. (1978). Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274.
- Wilson, A. and Manocha, D. (2003). Simplifying complex environments using incremental textured depth meshes. *Proceedings of ACM SIGGRAPH/ACM Trans. on Graphics*, 22(3):678–688.

- Wilson, A., Mayer-Patel, K., and Manocha, D. (2001). Spatially-encoded far-field representations for interactive walkthroughs. *Proc. of ACM Multimedia*.
- Wonka, P., Wimmer, M., and Schmalstieg, D. (2000). Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82.
- Wonka, P., Wimmer, M., and Sillion, F. (2001). Instant visibility. In *Proc. of Eurographics*.
- Woo, A., Poulin, P., and Fournier, A. (1990). A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32.
- Xia, J. C. and Varshney, A. (1996). Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96*. IEEE. ISBN 0-89791-864-9.
- Yoon, S., Salomon, B., and Manocha, D. (2003). Interactive view-dependent rendering with conservative occlusion culling in complex environments. Technical Report TR03-015, Department of Computer Science, University of North Carolina. Appeared in Proc. of IEEE Visualization 2003.
- Zhang, H., Manocha, D., Hudson, T., and Hoff, K. (1997a). Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*.
- Zhang, H., Manocha, D., Hudson, T., and III, K. E. H. (1997b). Visibility culling using hierarchical occlusion maps. Annual Conference Series, pages 77–88. ACM SIGGRAPH, ACM Press.